THE UNIVERSITY
of EDINBURGH

Efficient and Portable Multi-Tasking for Heterogeneous Systems

Christos Margiolas

Doctor of Philosophy

Institute of Computing Systems Architecture

School of Informatics

University of Edinburgh

2015

Abstract

Modern computing systems comprise heterogeneous designs which combine multiple and diverse architectures on a single system. These designs provide potentials for high performance under reduced power requirements but require advanced resource management and workload scheduling across the available processors.

Programmability frameworks, such as OpenCL and CUDA, enable resource management and workload scheduling on heterogeneous systems. These frameworks fully assign the control of resource allocation and scheduling to the application. This design sufficiently serves the needs of dedicated application systems but introduces significant challenges for multi-tasking environments where multiple users and applications compete for access to system resources.

This thesis considers these challenges and presents three major contributions that enable efficient multi-tasking on heterogeneous systems. The presented contributions are compatible with existing systems, remain portable across vendors and do not require application changes or recompilation.

The first contribution of this thesis is an optimization technique that reduces host-device communication overhead for OpenCL applications. It does this without modification or recompilation of the application source code and is portable across platforms. This work enables efficiency and performance improvements for diverse application workloads found on multi-tasking systems.

The second contribution is the design and implementation of a secure, user-space virtualization layer that integrates the accelerator resources of a system with the standard multi-tasking and user-space virtualization facilities of the commodity Linux OS. It enables fine-grained sharing of mixed-vendor accelerator resources and targets heterogeneous systems found in data center nodes and requires no modification to the OS, OpenCL or application.

Lastly, the third contribution is a technique and software infrastructure that enable resource sharing control on accelerators, while supporting software managed scheduling on accelerators. The infrastructure remains transparent to existing systems and applications and requires no modifications or recompilation. In enforces fair accelerator sharing which is required for multi-tasking purposes.

Lay Summary of Thesis

Modern computing systems comprise heterogeneous designs which combine multiple and diverse architectures on a single system. These designs provide potentials for high performance under reduced power requirements but require advanced resource management and workload scheduling across the available processors.

This thesis considers these challenges and presents three major contributions that enable efficient multi-tasking on heterogeneous systems. The presented contributions are compatible with existing systems, remain portable across vendors and do not require application changes or recompilation.

Acknowledgements

First of all, I would like to thank my academic advisor, Professor Michael O'Boyle, for his support over the past years. His advice and guidance have been invaluable both for pursuing my PhD and a successful career.

Next, I would like to thank my friends and colleagues in the CArD research group for our technical discussions, continuous brainstorming and funny moments. In particular, I would like to thank Alberto Magni, Konstantina Mitropoulou, and Vasileios Porpodas who helped me understand the challenges of academia during the first months of my studies. I also thank my officemates Harry Wagstaff, Stephen Kyle, Thibaut Lutz, Thomas Spink and Tobias Edler von Koch. I am also grateful to my good friends and colleagues Chris Fensch, Juan José Fumero, Kiran Chandramohan, Yuan Wen, Stavros Gerakaris and Dimitrios Papadopoulos. I would also like to thank Stratis Viglas for our Big Data and startup discussions.

I would like to thank my old buddies that we are still in touch and share technical ideas. In particular, I thank Anastasios Panagianis, George Kiagiadakis, George Kafentzis, Georgios Detorakis, Ioannis Manousakis, Michael Alvanos and Stathis Zavvos.

In addition, I would like to thank Bob Dreyer, Jason Kim and Mike Sharp for nine insightful and challenging months at the Qualcomm Innovation Center.

I thank my Bay Area friends Adam Smith, Behrooz Mahasseni, Gokul Subramanian, Hariharan Bhagavatheeswaran, Lee Yuan and Dimitrios Skarlatos.

As an act of narcissism, I would also like to thank myself for enduring the academic procedures and investing infinite hours in interesting projects.

Finally, I would like to thank my family for their unconditional support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

Some of the material used in this thesis has been published in the following papers:

- Christos Margiolas and Michael F. P. O'Boyle. *"Hyda: A hybrid dependence analysis for the adaptive optimisation of opencl kernels"*. In Proceedings of the International Workshop on Adaptive Self-tuning Computing Systems, ADAPT 2014.

- Christos Margiolas and Michael F. P. O'Boyle. *"Portable and transparent host-device communication optimization for gpgpu environments"*. In Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014.

- Christos Margiolas and Michael F.P. O'Boyle. *"Palmos: A transparent, multi-tasking acceleration layer for parallel heterogeneous systems"*. In Proceedings of the 29th ACM on International Conference on Supercomputing, ICS 2015.

- Christos Margiolas and Michael F.P. O'Boyle. *"accelOS: Portable and transparent software managed scheduling on accelerators for fair resource sharing"*. Under submission.

(*Christos Margiolas*)

Table of Contents

1 Introduction **1**

1.1 Multi-tasking Challenges on Heterogeneous Systems 2

1.2 Contributions 3

1.3 Thesis Outline 4

2 Technical Background **7**

2.1 Heterogeneous Systems 7

2.1.1 Concept 7

2.1.2 Central Processing Units 9

2.1.3 Graphics Processing Units 9

2.1.4 Non-Uniform Memory Access Systems 12

2.1.5 Thesis Directions 13

2.2 Accelerator Programmability 14

2.2.1 Concept 15

2.2.2 OpenCL 16

2.2.3 CUDA Comparison 20

2.2.4 Thesis Directions 21

2.3 Runtime Environments 22

2.3.1 Concept 22

2.3.2 Design Aspects 23

2.4 Compiler Infrastructure 24

2.4.1 Concept 24

2.4.2 LLVM Compiler Infrastructure 25

2.5 Evaluation Methodology 27

2.5.1 Metrics 27

2.5.2 Benchmarks 28

2.6 Summary 28

3 Related Work **31**
3.1 System Resource Sharing . . . 31
3.1.1 Homogeneous Systems . . . 32
3.1.2 Heterogeneous Systems . . . 33
3.1.3 Inter-Node Accelerator Resource Sharing . . . 35
3.2 System Resource Virtualization . . . 36
3.2.1 Hypervisor based Virtualization . . . 36
3.2.2 OS-level Virtualization . . . 38
3.2.3 Accelerator Virtualization . . . 39
3.3 Workload Scheduling on Heterogeneous Systems . . . 40
3.3.1 Runtime and Compiler Approaches . . . 41
3.3.2 Computer Architecture . . . 42
3.4 Memory Management and Data Sharing . . . 44
3.4.1 Memory Allocators . . . 44
3.4.2 Non-Uniform Memory Access Architectures . . . 46
3.4.3 Communication Optimizations for Heterogeneous Systems . . . 46
3.5 Performance Evaluation & Modeling on Accelerators . . . 48
3.6 Summary . . . 50

4 Host-Device Communication Optimization **51**
4.1 Introduction . . . 52
4.2 Motivation . . . 53
4.2.1 Performance Impact . . . 56
4.2.2 Summary . . . 56
4.3 Optimization Overview . . . 57
4.3.1 Platform Characterization . . . 58
4.3.2 Application Characterization . . . 58
4.3.3 Runtime Optimization . . . 59
4.4 Platform Characterization . . . 59
4.4.1 Memory Allocation Policies . . . 59
4.4.2 Platform Characterization Procedure . . . 61
4.5 Application Tracing . . . 62
4.5.1 Call Trace . . . 63
4.5.2 Trace Compression . . . 65
4.6 Application Analysis . . . 66

4.7 Runtime Optimization . . . 67
4.7.1 Memory Allocation Manager . . . 68
4.8 Experimental Setup . . . 69
4.8.1 Platforms . . . 69
4.8.2 Benchmarks . . . 69
4.9 Results . . . 70
4.9.1 Results on NVIDIA GTX 580 . . . 70
4.9.2 Results on AMD Radeon HD 5970 and NVIDIA Tesla k20c . . . 75
4.9.3 Tuned Version of Parboil for NVIDIA . . . 75
4.9.4 What policy to use . . . 76
4.9.5 Comparing against a naive approach . . . 79
4.10 Summary . . . 79

5 Heterogeneous Acceleration Layer **81**
5.1 Introduction . . . 81
5.2 Motivation . . . 83
5.3 Layer Overview . . . 84
5.3.1 Key Design Choices . . . 85
5.3.2 PALMOS Structured Design . . . 86
5.4 Virtual OpenCL . . . 87
5.4.1 Shared Stack . . . 89
5.4.2 Shared Data . . . 89
5.5 Inter-space Memory Allocator . . . 90
5.5.1 Two-Level Memory Allocator . . . 90
5.5.2 Address Space Translator . . . 90
5.5.3 Lock-free Design . . . 92
5.6 Resource Manager & Application Scheduler . . . 92
5.6.1 PALMOS Session . . . 92
5.6.2 Application Scheduling . . . 94
5.6.3 NUMA Awareness . . . 96
5.7 Security . . . 96
5.8 Experimental Setup . . . 97
5.8.1 Workloads . . . 98
5.8.2 Platform . . . 98
5.8.3 Comparison to existing approaches . . . 99

5.9 Results . . . 100
5.9.1 Single application performance . . . 100
5.9.2 Multi-program performance . . . 105
5.9.3 PALMOS against existing approaches . . . 106
5.10 Summary . . . 108

6 Resource Sharing Control on Accelerators 111
6.1 Introduction . . . 111
6.2 Motivation . . . 115
6.2.1 Motivational Example . . . 115
6.2.2 Standard Scheduling Approach . . . 117
6.2.3 accelOS: Software Scheduling & Resource Sharing Control . 117
6.3 Accelerator Resource Sharing Scheme . . . 119
6.4 Infrastructure Overview . . . 121
6.5 Host Runtime . . . 122
6.5.1 Application Monitor . . . 122
6.5.2 Kernel Scheduler . . . 123
6.6 Just In Time Compilation . . . 123
6.6.1 Compilation Procedure . . . 124
6.6.2 Transformation Overview . . . 124
6.6.3 GPU Runtime Library . . . 127
6.6.4 Adaptive Scheduling . . . 127
6.7 Experimental Setup . . . 128
6.7.1 Evaluation Platforms . . . 128
6.7.2 Workloads . . . 128
6.7.3 Comparison to other approaches . . . 128
6.7.4 Metrics . . . 128
6.8 Results . . . 129
6.8.1 Fairness in Accelerator Sharing . . . 130
6.8.2 Concurrent Kernel Executions . . . 134
6.8.3 System Throughput . . . 135
6.8.4 accelOS Overhead . . . 136
6.8.5 Additional Evaluation Metrics . . . 139
6.9 Summary . . . 139

7 Conclusion **141**
7.1 Contributions 141
7.1.1 Host-Device Communication Optimization 141
7.1.2 Heterogeneous Acceleration Layer 142
7.1.3 Resource Sharing Control on Accelerators 142
7.2 Critical Analysis 142
7.2.1 Alternative Designs in Kernel Space 143
7.2.2 Feedback Driven Resource Management 143
7.2.3 Unified Management of Computation and Graphics Workloads 144
7.2.4 Performance Evaluation with non GPU accelerators 144
7.3 Future Work 144
7.3.1 Unified Management of Computation and Graphics 144
7.3.2 Workload Migration across Processors 145
7.3.3 Dynamic Code Optimizations 145
7.3.4 Power Aware Resource Management 145
7.3.5 Integrated and Mobile GPUs 145
7.3.6 Operating Systems running on Accelerators 145
7.4 Summary 146

Bibliography **147**

Chapter 1

Introduction

Modern computing systems increasingly have heterogeneous designs where multiple, diverse processors are combined on a single system. Such systems have the potential for high computational throughput with reduced power requirements. Processor heterogeneity is the evolution of multi-core Central Processing Unit (CPU) designs which replaced the high frequency and power greedy sequential processor architectures in mid 2000's.

A heterogeneous system comprises multiple processors with diverse architectures specialized to different types of computation. In these systems, appropriate resource management and workload scheduling deliver high performance while preserving low power requirements. The vast majority of modern systems ranging from embedded and mobile areas to data centers and High Performance Computing (HPC) have heterogeneous designs. They consist of multi-core CPUs and one or more computational accelerators. Accelerators, traditionally, were specialized and expensive processor designs with limited application areas. However, the Cell BE [86] architecture and later the evolution of Graphics Processing Units (GPUs) from fixed pipeline graphic processors to fully programmable multi-cores enabled general purpose programming on accelerators. Nowadays, the most popular family of accelerators are GPUs while Digital Signal Processors (DSPs) and Xeon Phi [88] serve special application needs in mobile systems and HPC, respectively.

Software development on heterogeneous systems introduces additional complexity due to the design of mainstream programming languages that exclusively assume general purpose CPUs. This has lead to the development of specialized languages and programming frameworks, such as OpenCL[58] and CUDA[81], which enable work-

load scheduling on accelerator resources. These frameworks fully assign the control of resource allocation, data communication and workload scheduling to the application and there is no centralized control for resource management and scheduling. This design is sufficient for dedicated application systems but introduces significant challenges for multi-tasking environments where multiple applications and users compete for accessing system resources. This thesis addresses these challenges and provides solutions that enable efficient multi-tasking on existing hardware and software stacks without requiring any modification or recompilation of existing applications or any software stack changes.

1.1 Multi-tasking Challenges on Heterogeneous Systems

The existing software stack and programming models of heterogeneous systems lack efficient support for multi-tasking. They provide full control of accelerator resources to user applications and there is no central management. This leads to significant performance overhead, inefficient resource management and unfair accelerator sharing. The following paragraphs present some of the key challenges for multi-tasking on heterogeneous systems.

Communication Overhead: Modern heterogeneous systems consist of multiple processors which may have separate physical memories. This design guarantees high throughput memory accesses for the processors but introduces data copies across the different memories. These data copies cause performance overhead as they are not part of the original application payload but data communication enforced by the architecture design. Communication overhead may severely reduce the benefits of leveraging accelerators and are well known performance bottlenecks across all the types of heterogeneous software including applications running on HPC environments and multi-tasking systems. In particular, applications running on multi-tasking systems present relatively small workloads and communication overheads may dominate application execution times.

Vendor Interoperability: Programming accelerators requires new programming models and a number of software stack components including runtime environments, Just In Time (JIT) compilers and kernel drivers. Each vendor tends to provide its own software stack design which is typically closed source and incompatible with

third party designs. This is the case even for implementations of OpenCL, which is a standard programming model for leveraging accelerators in a portable manner. This prevents efficient use of multi-vendor heterogeneous systems and hinders the design of unified software stacks that are portable across vendors.

Accelerator Management: The existing programming models provide full control of accelerators to applications and there is no central management of accelerator resources. An application fully controls accelerator allocation, data transfers and workload scheduling. This design is sufficient for dedicated application systems where a single application exclusively executes on the system. In contrast, this design is problematic for multi-tasking systems. The lack of central management leads to unbalanced system sharing, poor resource allocation decisions and significant performance overheads. Furthermore, the lack of vendor interoperability, as described above, complicates the management of multi-vendor heterogeneous systems.

Accelerator Resource Sharing: The existing programming models and software stacks do not provide resource sharing control on accelerators for parallel execution requests made by different applications and users. A high level abstraction of accelerator resources is available to the developer and the application but there is no mechanism to control resource allocation. On modern dedicated GPUs, where context switch is not supported, the first application that performs a request may exclusively reserve accelerator resources. This behavior leads to scenarios where one application dominates accelerator usage, while other applications may suffer long delays before performing their computations on accelerators. This unfair accelerator sharing and the limited opportunities for concurrent accelerator usage undermines the capabilities of a multi-tasking environments. Some users or applications are favored while others suffer long delays.

1.2 Contributions

This thesis presents state of the art solutions to the four challenges described above. An automatic technique that optimizes host-device communication is proposed which reduces communication overheads and improves application performance. A virtualization layer for heterogeneous resources is then presented. It enables central management of accelerators, inter-vendor accelerator sharing and improves both application and system performance. It solves the vendor interoperability and accelerator management challenges described above. Finally, a technique that enables resource sharing

control on accelerator is proposed.

The following list briefly summarizes the main contributions of this thesis:

- The first contribution of this thesis develops an approach that reduces host-device communication overhead for OpenCL applications. It does this without modification or recompilation of the application source code and is portable across platforms. It achieves this by tracing and analyzing calls to the runtime made by the application and then selecting the best platform specific memory allocation and communication policy. It delivers speedups for a large number of applications. A detailed description is given in chapter 4.

- The second contribution is the design and implementation of a secure, user-space virtualization layer that integrates the accelerator resources of a system with the standard multi-tasking and user-space virtualization facilities of the commodity Linux OS. It targets heterogeneous commodity systems found in data center nodes and requires no modification to the OS, OpenCL or application. It eliminates high setup overhead, enables fine-grained sharing of mixed-vendor accelerator resources and provides resource and platform aware scheduling. It delivers application speedups and system throughput speedups. This work is presented in chapter 5.

- The last major contribution is a technique and a runtime and Just In Time compiler infrastructure that enable resource sharing control on accelerators, while also enabling software managed scheduling on accelerators. The infrastructure remains transparent to existing systems and applications and requires no modifications or recompilation. It delivers fairness improvements, system throughput speedups and application speedups. Chapter 6 presents this contribution in more detail.

1.3 Thesis Outline

The remainder of the thesis is organized as follows:

Chapter 2: This chapter presents the technical background that is required for the understanding of this thesis. It first introduces the concept of heterogeneous systems and discusses accelerator programmability. It then describes key concepts on runtime environments and compiler infrastructures. It concludes by presenting the evaluation methodology used in this thesis.

Chapter 3: This chapter discusses prior work. It first presents research in the areas of resource sharing and virtualization. It then introduces workload scheduling and communication optimization techniques. It concludes by reviewing memory management and performance evaluation techniques.

Chapter 4: This chapter presents an approach that reduces host-device communication overhead for OpenCL applications. It does this without modification or recompilation of the application source code and is portable across platforms. This approach outperforms competitive approaches. This chapter is based on the work published in paper [69].

Chapter 5: This chapter introduces a secure, user-space virtualization layer that integrates the accelerator resources of a system with the standard multi-tasking and user-space virtualization facilities of commodity Linux OS. The infrastructure remains transparent to existing systems and applications and requires no modifications or recompilation. The approach is evaluated on a large set of benchmarks and compared with alternative schemes. This chapter is based on the work published in paper [71].

Chapter 6: The chapter presents a technique and an infrastructure that enable resource sharing control on accelerators. The infrastructure remains transparent to existing systems and applications and requires no modifications or recompilation. The approach is evaluated on a large set of benchmarks and compared with alternative schemes. This chapter is based on the work that is under submission in paper [70].

Chapter 7: This chapter concludes this thesis with a summary of the main contributions. It provides a critical analysis of some of the technical aspects and discusses ideas for potential future work.

Chapter 2

Technical Background

This section provides a high level overview of the technical background that is required for the understanding of this thesis and its contributions. Key concepts such as heterogeneity, runtime environments and compiler infrastructure are discussed here. Section 2.1 introduces the concept of heterogeneous computing and presents the main processor types used in heterogeneous systems. Section 2.2 presents the programming model and frameworks required for computing on accelerators. Runtime environments, their role and core functionality are described in section 2.3. Compiler technologies and the LLVM compiler infrastructure are discussed in section 2.4. Finally, common evaluation methodologies used to evaluate the contributions of this thesis are given in section 2.5. The chapter concludes with a summary in section 2.6.

2.1 Heterogeneous Systems

This section presents the key features of heterogeneous systems. It first introduces the heterogeneity concept by providing an abstract architecture and then describes popular processor families found in heterogeneous system configurations.

2.1.1 Concept

Heterogeneity refers to computer architectures where a system consists of more than one processor types. This type of systems typically combine diverse processor types that serve different performance needs. Efficient workload scheduling across the available processors can deliver high computational performance with reduced power requirements. Figure 2.1 shows an abstract representation of a heterogeneous system

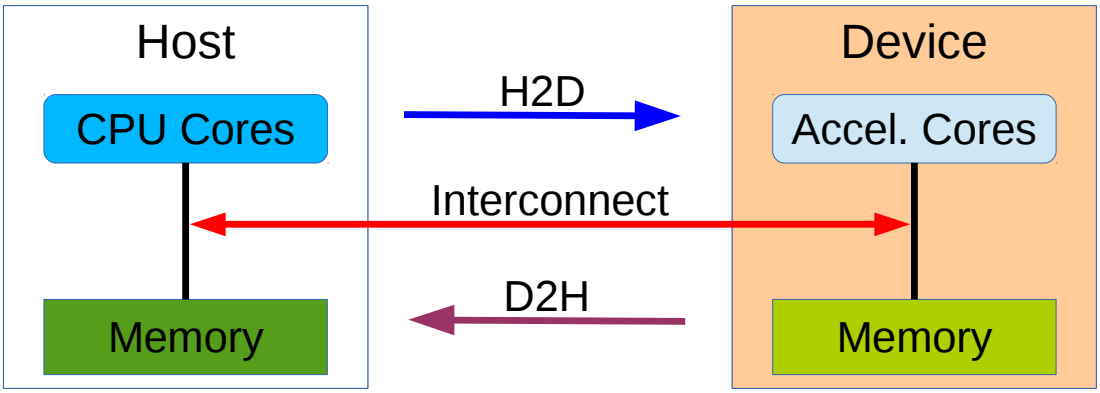


Figure 2.1: High level abstraction of a Heterogeneous System. It consists of two parts, the *host* and *device*. The host contains multi-core CPUs and the main system memory, while the device contains accelerator processors and their local memory sub-systems. An interconnect supports data communication in both directions.

that consists of two processor types. The first processor is a multi-core CPU which along with the main system memory are known as the *host* . The second processor is a computational accelerator, such as Graphics Processing Units (GPUs), which has a separate memory subsystem. The accelerator and its local memory are known as the *device*. The host and device are connected via an interconnect and data communication can take place in both directions. The operation of copying data from host memory to the device memory is named *Host to Device Communication (H2D Communication)*, while data copying to the opposite direction is named *Device to Host Communication (D2H Communication)*.

In modern systems, the host processor is frequently of x86 or ARM architecture. While the accelerator processor on the device may be a GPU, a Digital Signal Processor (DSP), a Field-programmable gate array (FPGA) or Intel Xeon Phi [88]. However, heterogeneous computing evolves rapidly and new vendors and products may provide divergent solutions.

There are heterogeneous system designs where different types of processing cores share the same chip and a single physical memory sub-system. This approach is followed by Intel processors with integrated graphics and the AMD Accelerated Processing Units (APU). In addition, single-ISA heterogeneous chips, such as the ARM big.LITTLE architecture, fall into this category. This type of chips is equipped with multiple types of cores that share the same instruction set but have different computing and power capabilities.

This abstract heterogeneous system definition can be generalized to system con-

figurations where the number and type of accelerators may vary. The next sections present the key features of CPU and GPU architectures which are components of the heterogeneous systems considered in this thesis.

2.1.2 Central Processing Units

The Central Processing Unit (CPU) is the de-facto processor type used in every modern computing system. It is a general purpose processor that performs all the computations required by any level of software including the Operating System and user applications. Over the years CPU architectures have evolved from sequential designs to multi-core, parallel processors where a single chip die has multiple identical cores that perform computations in parallel.

CPUs may incorporate additional levels of parallelism such as superscalar designs and instruction pipelining which permit multiple instructions to execute concurrently. They may also support Single Instruction Multiple Data (SIMD) operations which enable the parallel processing of multiple data elements per single instruction. SIMD is typically supported via architecture extensions such as Streaming SIMD Extensions (SSE) version 4 and Advanced Vector Extensions (AVX) version 2 for x86 and Advanced SIMD extension (NEON) for ARM architectures.

A modern processor typically is equipped with at least three levels of memory caches in order to mitigate the large main memory access times. Every level has been designed with a trade-off between speed and capacity. Cache levels closer to the CPU components tend to be faster but smaller in capacity while levels closer to memory have slower access times but larger capacity. Depending on the CPU design and the type of memory coherency, a cache level may be shared or not among the chip cores and the cache coherence protocol is responsible for enforcing a consistent memory view across the cores.

2.1.3 Graphics Processing Units

A Graphics Processor Unit (GPU) is a processor type that originally was targeted of graphics computation and acceleration. The first generations of GPUs were processors with fixed-function hardware units that performed predefined graphics computations. However, over the last 15 years GPUs have evolved to powerful data parallel architectures that can perform both graphics and general purpose computations. They contain a large number of programmable cores that are capable of performing complex graph-

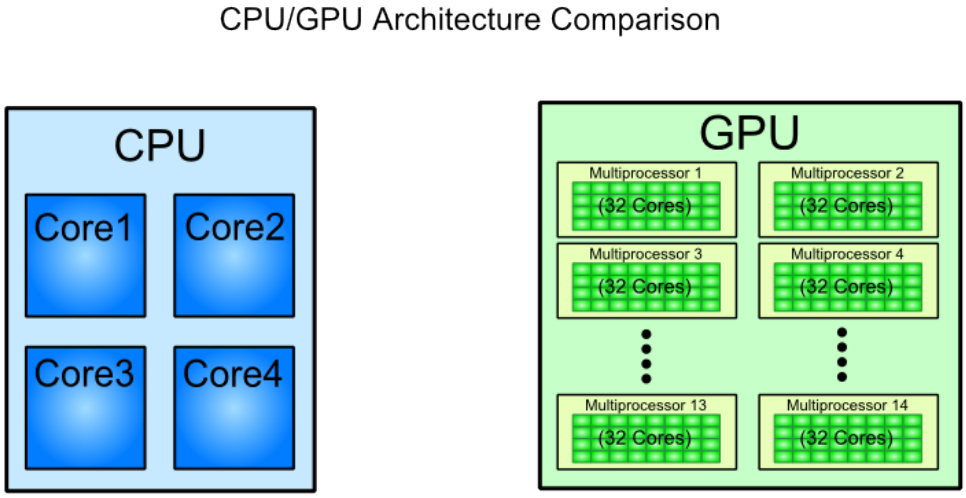


Figure 2.2: Abstract comparison of CPU and GPU architectures. CPU designs contains a small number of powerful cores while GPU designs contains thousands basic cores. CPUs efficiently handle irregular program control flows and memory accesses while GPUs specialize in data parallel workloads.

ics computations which are described in software. This flexible scheme also allows general purpose computing on GPUs for data parallel applications. Application developers use programming models such as OpenCL and CUDA and write general purpose tasks that are executed on GPUs. This type of programming is called *General-purpose computing on graphics processing units (GPGPU)*.

GPU architecture designs are radically different to CPUs as can be seen in figure 2.2. While CPUs typically have a small number of powerful cores and advanced memory caches, GPUs follow a different approach. GPUs have hundreds of small computational cores which are efficient for computation operations but they lack efficiency for managing complex program control flows and irregular memory access patterns. Furthermore, GPU designs either completely discard cache use or they provide a very basic cache hierarchy with low space and logic requirements.

Multiprocessors: GPUs have a massive number of cores which are grouped in *Multiprocessors*. The cores of a multiprocessor share a single Program Counter and they work in lockstep; they perform computations in a Single Instruction Multiple Data (SIMD) manner. Every core executes the same instruction at a time but on different data. This design is extremely effective for computation operations but introduces significant overhead for divergent control flows. Each time a branch instruction sets different execution paths across the cores, the execution of the different paths is per-

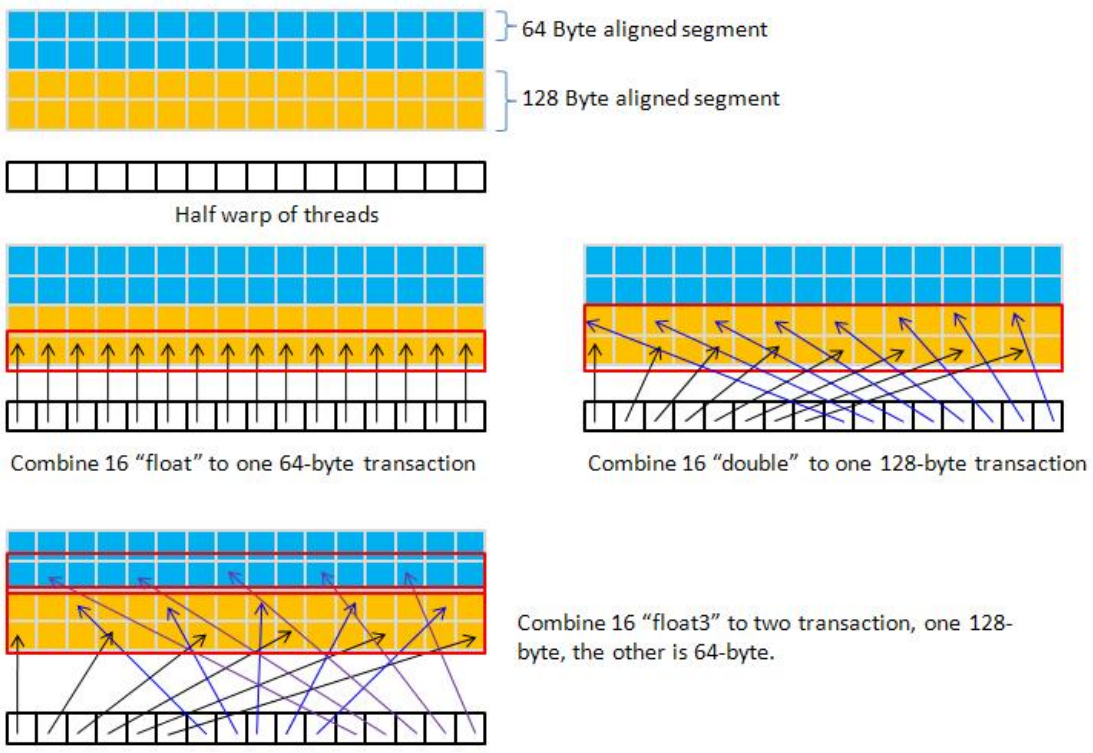


Figure 2.3: Memory Access Coalescing on GPU architectures. In order to achieve peak performance, the cores of a multiprocessor are required to access consecutive addresses in memory. Figure taken from [81].

formed sequentially. Furthermore, the cores of a multiprocessor share a single large Register file and a software managed scratchpad memory which is typically named *local memory* and is accessible by the programming models. A GPU contains multiple multiprocessors and each of which may execute different parts of a program or even different programs. This way Single Program Multiple Data and Multiple Program Multiple Data models are supported by GPUs.

Memory Hierarchy: GPUs originally did not have any memory cache. Latest generations come with two levels of cache. The first level, L1, is typically shared among the cores of a multiprocessor, while the second level, L2, is shared among all the cores of the GPU. L1 cache and local memory share the same silicon area and their capacity is adjustable. A GPU has access to a *global memory* that is based on DRAM technology. The global memory may be a dedicated memory subsystem or the main system memory in case of integrated GPUs. GPU memory access interfaces are wider than standard CPU interfaces and are meant to serve multiple cores per memory read or write request. These wide interfaces provide higher memory access throughput but

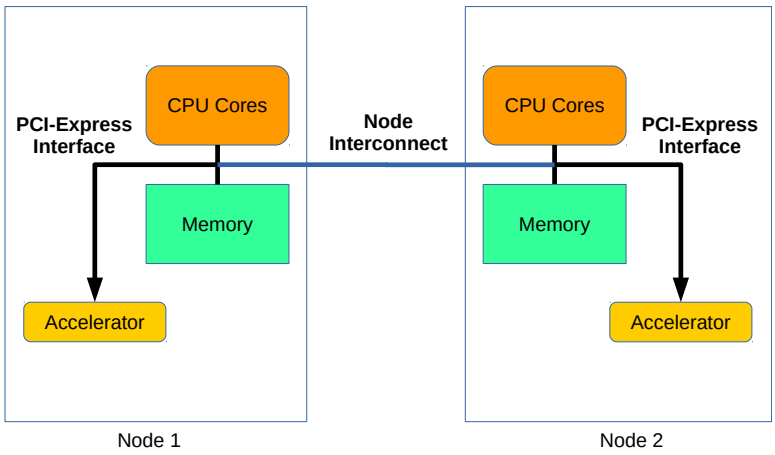


Figure 2.4: Abstraction of a Non-Uniform Memory Access (NUMA) System. CPU chips and physical memory are distributed in multiple nodes (or sockets). System nodes are connected via an interconnect.

introduce a significant constraint. In order to achieve peak performance, the cores of a multiprocessor are required to access consecutive addresses in memory as it is shown in figure 2.3. This type of memory accesses is called *memory coalescing*. Depending on the architecture capabilities different memory access patterns are supported for memory coalescing.

2.1.4 Non-Uniform Memory Access Systems

Non-Uniform Memory Access (NUMA) Systems are a family of shared memory architectures where CPU chips and physical memory are distributed in multiple nodes (or sockets). As can be seen in figure 2.4, every node typically consists of a local CPU socket and local memory slots, while the nodes are connected and communicate via an interconnect. The key feature of this design is that intra-node memory accesses are faster than inter-node accesses. Accessing local memory only involves communication between a local CPU core and the local memory subsystem while, while inter-node communication requires additional communication over the interconnect. The abstract heterogeneous system presented here also presumes that every node has a local PCI Express interface and that an accelerator is connected on it. Whilst, this type of systems supports highly scalable shared memory systems it also introduces challenges in memory management and thread scheduling.

Thread Mapping and Data Placement: Due to the distributed design of NUMA systems, thread scheduling and memory management should be aware of system topol-

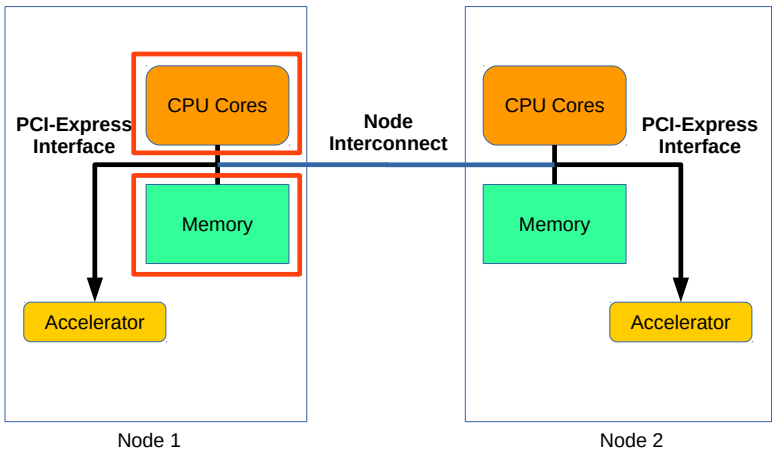


Figure 2.5: Efficient Thread Mapping and Data Placement on NUMA systems. An application using CPU cores and memory from a single node suffers minimal inter-node communication.

ogy and resource locality. A failure to detect and exploit resource locality leads to poor performance because of the introduced inter-node interconnect overhead. Ideally, the threads of a process should be mapped to a single node and allocate memory from the same node, as shown in figure 2.5. However, this is not always feasible, application requirements may exceed the node resources or the node resources may be reserved by other processes. In practice, an Operating System or a runtime resource manager performs best-effort thread scheduling and data placement.

Heterogeneity Challenges: Heterogeneous computing on NUMA systems introduces additional complexity. Performing computations on accelerators involves data transfers between the system and accelerator memories and requires extensive synchronization. Accelerator location should be considered in resource allocation decisions. An accelerator that is locally connected to the node where the host part of the application runs will deliver better performance than an accelerator connected to a remote node. Once, again, the threads of a process should be mapped to a single node and allocate memory from the same node and use the accelerator that is locally connected to the node, as it is shown in figure 2.6. However, an accelerator may not be available or be reserved by other processes and a system should follow a best-effort approach.

2.1.5 Thesis Directions

This thesis explores software multi-tasking on heterogeneous systems. It delivers state of the art solutions for multi-tasking via accelerator resource virtualization and re-

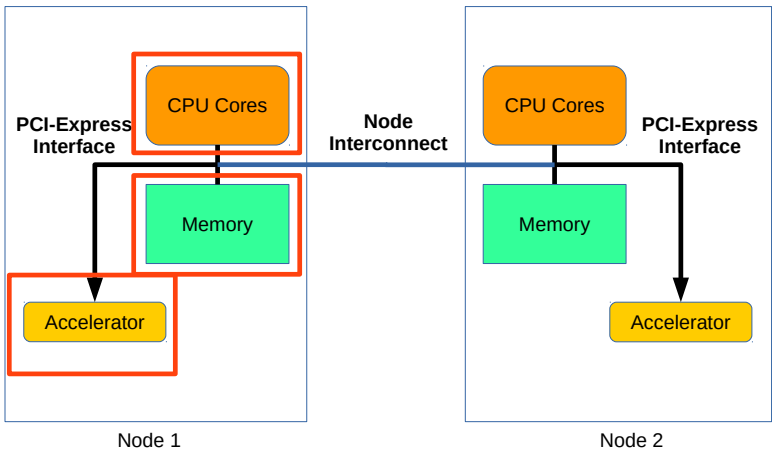


Figure 2.6: Efficient Thread Mapping, Data Placement and Accelerator Selection on NUMA systems. An application using CPU cores, memory and accelerators from a single node suffers minimal inter-node communication.

source sharing control on accelerators. Furthermore, it delivers significant performance improvements via a novel communication optimization and improved system resource management.

All the presented contributions are software solutions and do not require any modification of the application code, Operating System or runtime environments. This thesis work integrates seamlessly with the existing software stacks and does not require any hardware changes.

Key choice of this work is the development of portable designs which are applicable to a broad range of heterogeneous systems and accelerators types which have varying hardware capabilities. This thesis presents solutions involving the development of new runtime environments and compiler technologies that rely on open programming standards, such as OpenCL, for accessing and leveraging multi-vendor accelerator resources. The proposed designs, however, take advantage of the diverse underlying accelerator hardware and specialize their resource management and sharing practices.

2.2 Accelerator Programmability

This section provides a high level description of the programming models used for heterogeneous computing. It first introduces the workload offloading concept, the key operation for computing on accelerators. It then provides a detailed description of the OpenCL programming model which is an open standard that supports a large range of processor types including GPUs and CPU multi-cores. In conclusion, a short descrip-

tion of CUDA programming model, which targets NVIDIA GPUs, is given.

2.2.1 Concept

Workload offloading, shown in figure 2.7, is the key operation in heterogeneous computing which takes place every time an application performs tasks on accelerators. It is the procedure of assigning a task for computation to the device of a heterogeneous system. It involves three steps. It first copies the input data from host to the device memory, then dispatches the kernel for computation and the accelerator performs the computation. Finally, it copies the output data from device to the host memory. Data copies may not be necessary for heterogeneous systems that support shared memory. However, it may be still desirable for performance purposes.

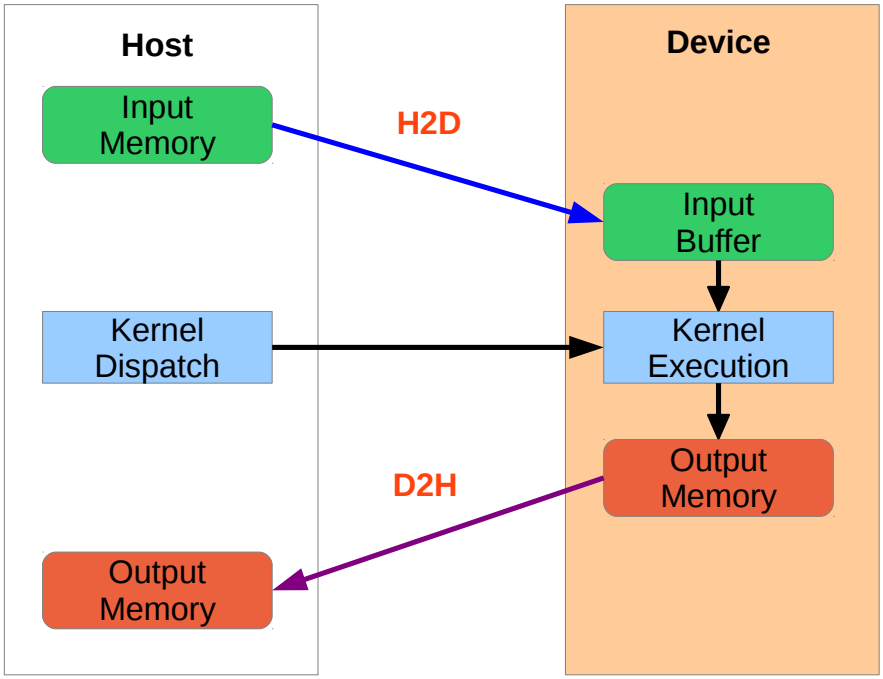


Figure 2.7: Workload Offloading for Computation on Accelerators. It first copies input data from host to the device memory, then dispatches a kernel for computation to the accelerator. The accelerator performs the computation and the output data is copied back to the host.

Runtime support is required for managing accelerator resources, controlling data communication and workload scheduling. Compiler infrastructure with Just In Time (JIT) capabilities is also required for handling multi-target compilations and performing dynamic code specialization. OpenCL and CUDA are two frameworks for hetero-

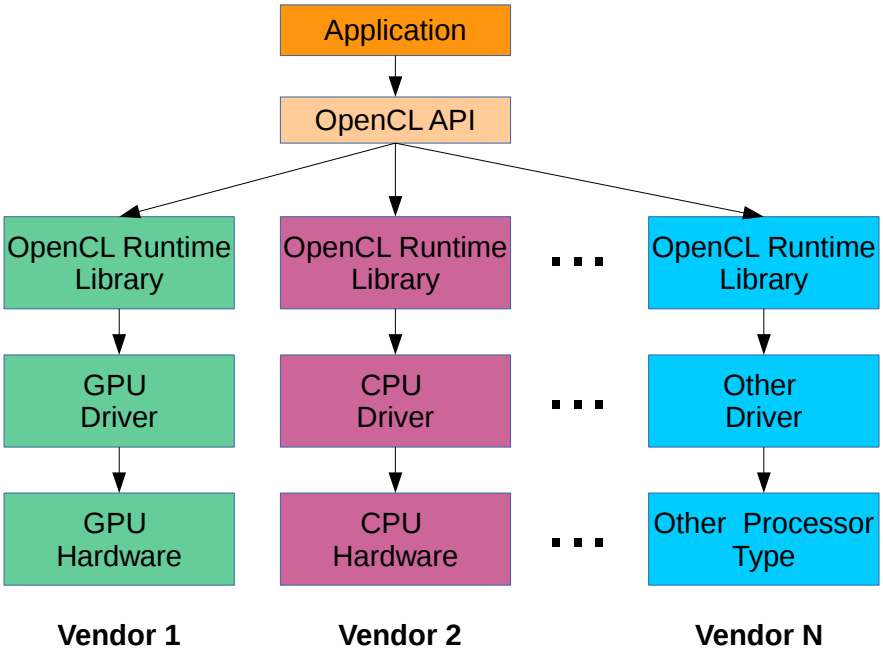


Figure 2.8: OpenCL Application Portability. An application may perform across diverse processor types. The OpenCL standard defines a library interface that is supported by multiple processor vendors. A single application implementation written in OpenCL can leverage all these processors.

geneous computing that provide runtime and compiler support and encapsulate workload offloading as part of a programming model.

2.2.2 OpenCL

OpenCL is an open standard that enables heterogeneous computing for a large range of processors including CPU multi-cores, GPUs, FPGAs and DSPs. OpenCL organizes a program code in two parts, the *host code* and *device code* which respectively run on the host and device of a heterogeneous system.

The OpenCL standard defines a host library and OpenCL C language. The library, which is linked against the host code, handles accelerator management, data communication, workload scheduling and drives a Just In Time (JIT) compiler which compiles device code for the accelerator architectures. OpenCL C is a variant of C language that supports the development of device code. The device code is organized in special functions, named *kernels*.

An OpenCL application that is fully compliant with the OpenCL standard can be executed on diverse heterogeneous systems with various accelerator types, given that

the platform vendors provide OpenCL support. This application portability is shown in figure 2.8. OpenCL applications use the OpenCL library interface to interact with the OpenCL infrastructure while they provide device code in OpenCL C. Each vendor implements the OpenCL standard by providing individual versions of OpenCL library implementations and required drivers. An application can use different accelerators by accessing different OpenCL library implementations.

Device Abstraction: The OpenCL standard supports a wide range of processors and introduces a high level *device abstraction*, shown in figure 2.9. This abstraction supports data parallel architectures, such as GPUs and commodity multi-core CPUs. The resources of a device are organized in groups, named *Compute Units*. A Compute Unit typically consists of multiple hardware cores/threads, named *Processing Elements*. Every Processing Element maintains its own private memory while it also shares a local memory with the Processing Elements of the Compute Unit it belongs. Process Elements have access to a *Global Device Memory* which typically has a large capacity via a *Global Data Cache*. A device may also have a faster read-only global memory, named *Constant Memory* while a special cache, named *Constant Data Cache* improves access times to it. The OpenCL memory model is relaxed and hardware support for memory coherence is not considered. The memory state visible to one Process Element is not guaranteed to be the same across elements. A consistent view of local memory can be enforced at Compute Unit level via *barrier* operations. Furthermore, atomic memory access operations guarantee serialized access to memory locations both in local and global memories.

Platform Abstraction: OpenCL considers multiple accelerator types that may coexist on a platform at the same type. It provides a platform abstraction, shown in figure 2.12 where there is a single host and multiple devices are available. The OpenCL library interfaces permit an application to query the available devices and their capabilities and then reserve the desired devices. An application can leverage multiple devices at the same time, even devices coming from different vendors.

OpenCL Kernels: OpenCL C is a subset of C99 language with extensions to support multiple address spaces, vector data types and textures. Device code is written in OpenCL C which comprises special functions named *OpenCL kernels*. A kernel example is given in figure 2.11. This kernel performs vector addition, it reads elements from A and B and writes the results on C. This kernel is executed in parallel across the Processing Elements of the device and special intrinsic functions, such as *get_global_id*, are required to index the appropriate data.

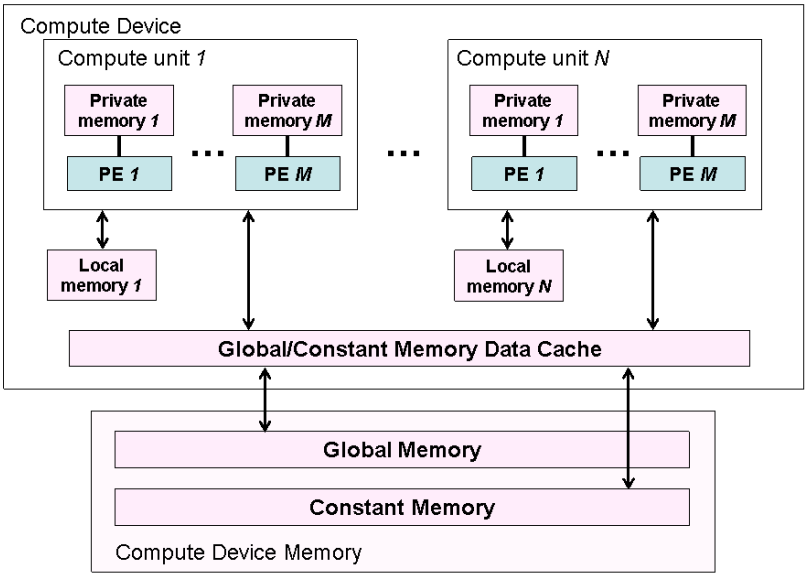


Figure 2.9: OpenCL Device Abstraction. The OpenCL environment provides an abstract representation of accelerator resources. This abstraction supports data parallel architectures, such as GPUs and commodity multi-core CPUs. Figure taken from[58].

Kernel Execution Space: A kernel execution launches multiple instances of the kernel code. These instances are organized within a multi-dimensional index-space, named *NDRange*, as shown in figure 2.12. Each instance of the kernel execution is named *work item*. Furthermore, work items are organized in groups, named *work groups*, where a work item is member of a single work group. The sizes of the execution space and work groups are specified during application runtime. Each work item is globally characterized by a unique tuple, named *global_id* and a second tuple, named *local_id*, which uniquely characterizes it within its work group. Work groups are characterized by a unique tuple, named *group_id*. During a kernel execution, the device scheduler first maps work groups to Compute Units and then work items to Processing Elements.

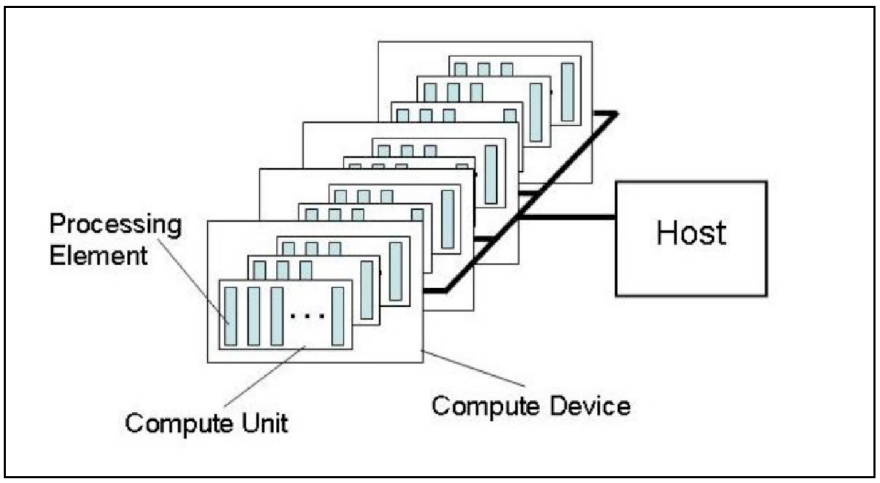


Figure 2.10: OpenCL Platform Abstraction. Figure taken from[58].

```
kernel void vectoradd(global float *A, global float *B,
                      global float *C)
{
  int i = get_global_id(0);
  C[i] = A[i] + B[i];
}
```

Figure 2.11: Example of an OpenCL kernel that performs vector addition.

The following terms describe a kernel execution space in OpenCL [58]:

- NDRange space size: (Gx,Gy)
- Size of each work-group: (Sx,Sy)
- Number of work-groups: (Wx,Wy)
- Work-item global indexes: (gx,gy)
- Work-group global indexes: (wx,wy)
- Work-item local indexes, inside the work-group: (lx,ly)

The above terms are correlated and the following formulas define their relationship:

- (gx,gy) = (wx Sx +lx,wy Sy +ly)
- (Wx,Wy)=(Gx/Sx,Gy/Sy)
- (wx,wy) = ((gx sx)/Sx,(gy sy)/Sy)

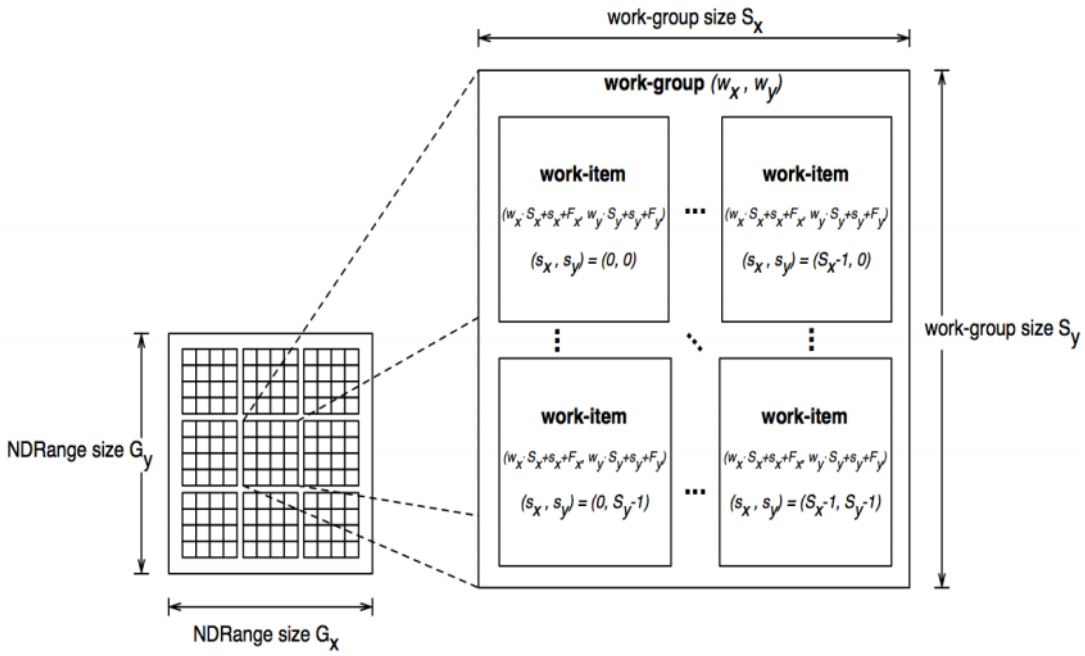


Figure 2.12: OpenCL Kernel Execution Space (NDRange). A kernel execution launches multiple instances of the kernel code. These instances are organized within a multi-dimensional index-space, named *NDRange*. Figure taken from[58].

Just In Time Compilation: OpenCL applications provide device code in OpenCL C or binary representations and the OpenCL runtime is responsible for driving a Just-In-Time compiler that compiles the device code for the target accelerator architectures. This is done in two steps as shown in figure 2.13. The application first provides the device code to the OpenCL runtime via a call to *clCreateProgramWithSource* or *clCreateProgramwithBinary* which generates a new clProgram object. The second step requires a call to *clBuildProgram* with the newly created clProgram. This call generates native code for the accelerator architecture. The OpenCL standard also defines low level interfaces that provide additional control over the compilation procedure.

2.2.3 CUDA Comparison

The CUDA framework is a proprietary software stack developed by NVIDIA that exclusively enables heterogeneous computing on NVIDIA GPUS. However, it was developed and released before the OpenCL framework and has been the pioneer in many aspects of heterogeneous computing, specially in the area of GPUs. OpenCL and CUDA share many concepts and have similar programming models. In this thesis,

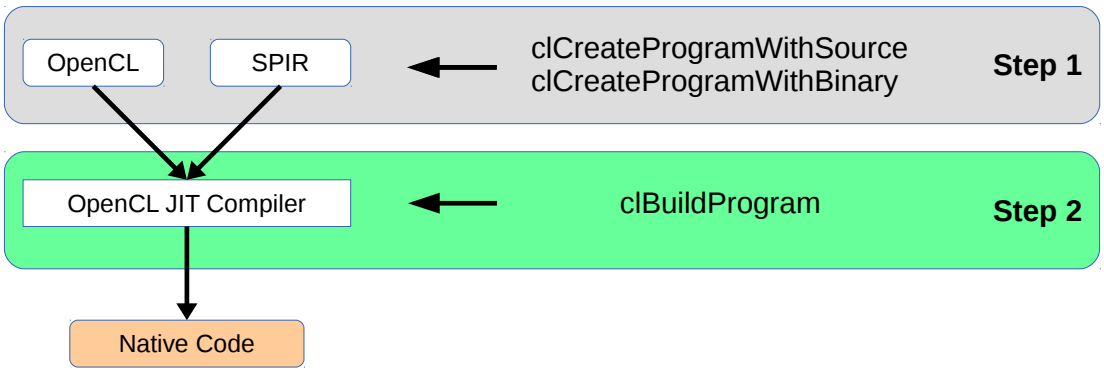


Figure 2.13: OpenCL Just In Time (JIT) Compilation. The OpenCL runtime performs a JIT compilation of the OpenCL kernels targeting the available accelerators.

OpenCL	CUDA
NDRange	grid
work-item	thread
work-group	thread block
global memory	global or device memory
constant memory	constant memory
local memory	shared memory
private memory	registers

Table 2.1: List of OpenCL terms and their equivalents in CUDA.

the OpenCL terminology has been adopted as it is broadly used and remains abstract and vendor independent. To understand how OpenCL and CUDA relate to each other, table 2.1 provides a list of key OpenCL terms and their equivalents in CUDA.

2.2.4 Thesis Directions

The work of this thesis relies on the Khronos OpenCL 1.x versions for accessing and utilizing accelerators. This enables immediate deployment for diverse accelerator types of different vendors. Furthermore, the presented compiler work also relies on the *Standard Portable Intermediate Representation (SPIR)* 1.x [59]. SPIR permits the use of vendor compiler backends as part of custom compilation infrastructures.

Recently released versions of OpenCL, such as OpenCL version 2.0 and 2.1, introduce new features such as shared virtual memory and dynamic parallelism. However, at the time of writing this thesis, no stable implementation of these specifications

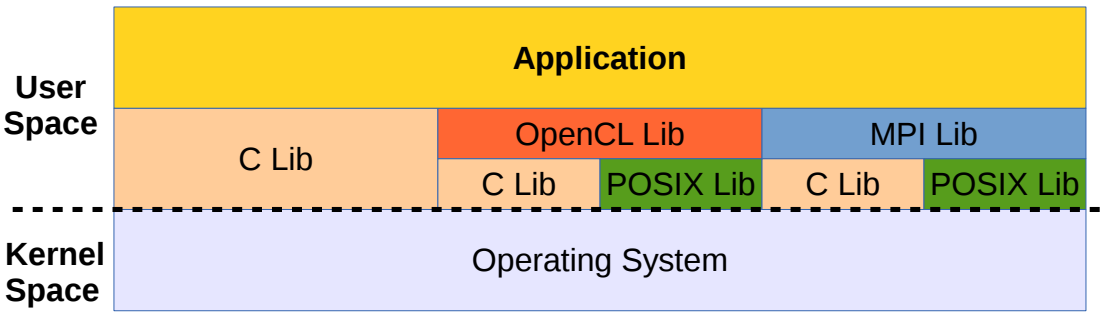


Figure 2.14: An application example that relies on multiple runtime environments. It relies on the C language runtime provided by C library, the OpenCL runtime provided by OpenCL library and the MPI runtime, provided by MPI library.

has been made available by hardware vendors. Furthermore, Khronos Vulcan, a new programming standard that unifies graphics processing and generic computation on accelerators, has been introduced. Again, at the time of writing this thesis no final specification or implementation is available.

The work and techniques presented in this thesis follow a generic approach which is directly applicable to future versions of OpenCL or other programming standards such as Khronos Vulcan. The virtualization, resource management and scheduling approaches can be extended to support new programming interfaces and leverage new hardware functionality.

2.3 Runtime Environments

This section provides a brief description of runtime environments. It first introduces the concept of runtime environment and then presents key design approaches.

2.3.1 Concept

A runtime environment is a software stack component that provides low level, system functionality to user applications or other runtime environments. The type of functionality delivered by a runtime environment is typically beneficial for a large number of applications. Typical examples of runtime environments are the C and C++ runtime libraries, networking libraries and the OpenCL library.

Figure 2.14 provides a high level visualization of a software stack where a user application relies on the functionality of a set of runtime environments for performing its tasks. In this example, we consider applications found in heterogeneous HPC

centers which leverage accelerators via OpenCL and rely on MPI[76] for inter-node communication and workload partitioning. The example application relies on the C language runtime provided by C library, the OpenCL runtime provided by OpenCL library and the MPI runtime, provided by MPI library. The C language runtime does not have any dependency on third runtimes and performs system calls to communicate with the kernel. Both OpenCL and MPI runtimes in turn rely on the C language and POSIX runtimes.

2.3.2 Design Aspects

This section discusses key aspects for the design and implementation of runtime environments. It first presents different runtime categories, then discusses system dependencies and finally compares the benefits of user-space development against kernel-space designs.

Runtime Environment Categories: Runtime environments are typically developed as system libraries where applications need to be statically or dynamically linked against them. An application then accesses the environment functionality by performing function calls to the library. However, other designs are available too. A runtime environment may be a system process or a kernel space task which is driven by system signals and interrupts or requests arriving from network sockets or Inter-Process shared memory. Language Virtual Machines is another popular category where the application binary targets an abstract architecture and the virtual machine is responsible for performing the application execution. Virtual machine operations require a complex runtime environment that manages memory allocations, task scheduling and code generation.

System Dependencies: The development of runtime environments typically requires access to system resource management. This is either done by relying on other runtime environments or, when required, by performing system calls directly to the kernel of the Operating System.

User-Space vs Kernel-Space: An important trade-off for designing new runtime environments is whether they should be developed in user-space or be part of the kernel components. Placing a runtime environment in user-space provides great flexibility and low overhead access to the runtime operations, while it comes with limited control and access to low system resource management which is performed by the Operating System kernel. Developing the new functionality in kernel-space provides full access

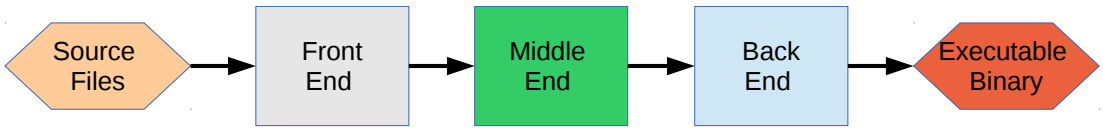


Figure 2.15: Overview of a Compilation Procedure. A modern compiler consists of three components, the front-end, middle-end and back-end.

to resource management but it introduces significant overheads for accessing its operations. In practice, user-space, kernel-space and hybrid approaches are in use. A typical case of hybrid designs is the OpenCL support on GPUs where parts of the runtime environments are developed in user-space while others are designed as part of kernel drivers.

2.4 Compiler Infrastructure

This section provides a brief description of compiler operations and then presents an overview of the LLVM compiler infrastructure which has been used in work presented in this thesis.

2.4.1 Concept

A compiler is a software program which is responsible for converting a software written in a source language such as C or C++ into target code which typically is an architecture computer language, known as *assembly*. It typically translates the source program into lower *intermediate representations* which are optimized before the target code generation. A modern compiler is organized in three components which are described in the next paragraphs.

Front-end: This compiler component is responsible for reading and analyzing the source code provided by the developer. It then generates an intermediate representation of the program that is semantically equivalent to the original source code. It operations involve a set of analyses phases. First, the *Lexical Analysis* organizes the program in valid tokens which are single atomic units of the language. Valid tokens are productions of regular expressions. *Syntax Analysis* uses the collected tokens and a *formal grammar* that defines the language in order to extract the syntactic structure of the program which is represented as *parse trees*. Then, *Semantic Analysis* takes place where the compiler checks the semantic correctness of the syntax trees and constructs *symbol*

tables. A preprocessing phase is also available for languages with preprocessing capabilities such as C and C++. Finally, if a source program has successfully passed all the described phases the front-end generates the intermediate representation.

Middle-end: This component reads the intermediate representation generated by the front-end and performs platform independent optimizations where it transforms the original intermediate representation to versions that are semantically equivalent but they have been altered to improve system performance. The optimizations are typically organized in independent passes which also require independent code analysis passes. Depending the compiler configurations the passes are organized in a pipeline and they are sequentially applied to the input intermediate representation. Common optimizations are Common Subexpression Elimination, Global Value Numbering, Dead Code Elimination, Constant Propagation, Loop Optimizations, Auto-Vectorization and Auto-Parallelization. The correctness of the code transformations is typically checked with rigorous testing. However there are alternative approaches where formal methods define the operation of transformations. Some compiler designs require the middle-end component to generate a different intermediate representation which is later used as input to the compiler Back-end.

Back-end: This component reads the intermediate code generated by the middle-end and transforms it to assembly language which is suitable for execution on the target architecture. Two major operations are performed by the compiler back-end, the *Register Allocation* and *Instruction Scheduling*. Modern intermediate representations assume an infinite number of registers and the back-end with its register allocation algorithm is responsible for mapping the registers of the intermediate representation to the actual registers of the processor. This procedure typically introduces the use of *program stack* due the limited number of physical registers of the processor. *Instruction Scheduling* is responsible for ordering the program instructions in a manner that maximizes processor performance while it does not violate their dependencies. Modern back-ends perform additional target specific optimizations.

2.4.2 LLVM Compiler Infrastructure

LLVM[63] is an open-source compiler infrastructure which provides a framework for aggressive code optimizations under reduced compilation times. LLVM follows a modular design where its components are provided as libraries which can be combined in different ways and serve varying application areas and performance goals. It

supports code generation for a large range of processor families ranging from mobile ARM to x86 and GPU architectures. Similarly LLVM supports multiple source languages and has Just In Time compilation capabilities. The next paragraphs present the key features and design of LLVM.

LLVM Design: LLVM follows a modular design where each compiler functionality is organized as a library which can be loaded as part of a compiler driver or directly used by an application. Furthermore, LLVM follows the three component compiler design we described in the previous section. The standard compiler front-end which supports C, C++ and their variations is named *clang* and is provided both as an end-user compiler driver and as a library for integration with external applications and compiler tools. The LLVM core libraries provide the functionality of the middle-end and an end-user utility, named *opt*, provides access to its functionality. Finally, LLVM has a set of libraries that support the back-end operations which perform code generation. The end-user utility that provides direct access to the back-end functionality is named *llc*.

LLVM Intermediate Representation: LLVM introduces a low level intermediate representation, named *LLVM IR*, that relies on Static Single Assignment (SSA)[90][24], where a variable declaration is assigned a value a single time. Programs in LLVM IR are organized into *Modules*. A module corresponds to a single translation unit of a program that has been processed by the front-end. Each module contains *Functions* and global variables and each function is made of *Basic Blocks*. A basic block is a set of *IR Instructions*. LLVM IR is an abstract assembly language that is independent of the source language and target architecture. It is portable across processor architectures and execution environments. Furthermore, it is extensible either via LLVM metadata or intrinsics.

Pass Manager: LLVM organizes optimizations and the required code analyses in *compiler passes*. LLVM supports multiple type of passes that may operate at module, function, basic block or loop level and provides mechanisms for expressing dependencies between the available passes. A *Pass Manager* is the mechanism that handles the scheduling of both analysis and optimization passes. It minimizes compilation times while respecting inter-pass dependencies.

OpenCL Transformations: Contributions of this thesis require the analysis and transformation of OpenCL kernel codes which are either provided in *OpenCL C* or *Standard Portable Intermediate Representation (SPIR)*[59]. The LLVM C/C++ front-end, clang, supports LLVM IR generation from OpenCL C files. SPIR is a subset of

LLVM IR and the existing LLVM infrastructure can correctly analyze and transform it. LLVM represents OpenCL concepts such as multiple address spaces, intrinsic function calls and kernel attributes via metadata and intrinsics. LLVM infrastructure supports code generation for both NVIDIA and AMD GPUs with experimental compiler back-ends and also provides a back-end for SPIR code generation.

2.5 Evaluation Methodology

The contributions of this thesis provide new functionality for heterogeneous computing and their performance evaluation is crucially important. This section presents the metrics and benchmarks used for the evaluation of the contributions.

2.5.1 Metrics

This section presents the performance evaluation metrics.

Relative Performance: The contributions of this thesis propose new software stack components that enable new functionality and improve performance. The performance improvements for individual tasks are quantified by the *speedup metric* which is defined as:

$$speedup = \frac{t_{baseline}}{t_{new}}$$

Where the $t_{baseline}$ is the execution time of the experiment running with the configuration considered as the *baseline*, while t_{new} is the execution time delivered by the contribution presented in this thesis. For system level evaluation, where the performance of a set of tasks is evaluated, the *throughput speedup* metric is used, which is defined as

$$throughputspeedup = \frac{T_{baseline}}{T_{new}}$$

Where $T_{baseline}$ is the total execution time for the experiment tasks running with the configuration considered as the baseline, while T_{new} is the total execution time delivered by the contribution of this thesis. Summary results are provided as the geometric mean of the speedups of the individual experiments.

Execution Time Breakdowns: Some evaluation scenarios presented in the next chapters require execution time breakdowns where the execution times of individual program phases are provided in addition to the total execution times. In those cases the

following condition is always valid:

$$\sum_{i=0}^{N} t_{phase_i} = t_{Total}$$

Where t_{phase_i} is the execution time for a particular phase of a program while t_{Total} is the total execution time of the program.

System Fairness: Chapter 6 evaluates system fairness. However because this type of evaluation is not performed in the rest of the thesis the required metrics are provided in the experimental setup of the chapter.

2.5.2 Benchmarks

The experiments performed in this thesis consider the Parboil[101] and Rodinia[18] benchmark suites. The exact configurations, benchmark selections and evaluation purposes are described in the experimental setup sections of the following chapters. A list of the available benchmarks is given in table 2.2.

2.6 Summary

This section presented the technical background required for the understanding of this thesis and its contributions. It first introduced the concept of heterogeneous computing and presented mainstream processor types used in heterogeneous systems. It then presented the programming model and frameworks required for computing on accelerators. Runtime environments, compiler technologies and the LLVM compiler infrastructure were briefly discussed. Finally, common evaluation methodologies used to evaluate the contributions of this thesis were given.

Before presenting the contributions of this thesis, starting in chapter 4, the following chapter discusses prior work relevant to this thesis.

Benchmark Name	Source
backprop	rodinia
bfs	parboil
bfs	rodinia
cfd	rodinia
cutcp	parboil
gaussian	rodinia
heardwall	rodinia
histo	parboil
hotspot	rodinia
kmeans	rodinia
lavaMD	rodinia
lbm	parboil
leukocyte	rodinia
lud	rodinia
mri-gridding	parboil
mri-q	parboil
nn	rodinia
nw	rodinia
particlefilter	rodinia
pathfinder	rodinia
sad	parboil
sgemm	parboil
spmv	parboil
srad	rodinia
stencil	parboil
streamcluster	rodinia
tpacf	parboil

Table 2.2: List of OpenCL benchmarks available in Parboil and Rodinia suites.

Chapter 3

Related Work

This chapter discusses prior work related to the areas covered in this thesis. A brief review of publications is provided for each area.

This chapter is organized as follows. Section 3.1 deals with resource sharing techniques and practices for multi-tasking systems where multiple users and applications compete for resource usage. Next, section 3.2 discusses virtualization technologies for homogeneous and heterogeneous systems. Prior work on workload scheduling on heterogeneous systems is discussed in section 3.3. Memory management and data sharing is discussed in section 3.4. This section specially discusses memory allocators, resource management for Non-Uniform Memory Access (NUMA) architectures and communication optimization techniques for heterogeneous systems. Prior work on performance evaluation and modeling on accelerators is presented in section 3.5. The chapter concludes with a brief summary of the discussed work in section 3.6.

3.1 System Resource Sharing

This section reviews prior work for resource sharing on multi-tasking systems where multiple users and applications compete for access to system resources. Section 3.1.1 presents related publications for homogeneous systems while section 3.1.2 discusses resource sharing on modern heterogeneous systems. Finally, section 3.1.3 reviews prior work related to accelerator sharing across multiple computing nodes that are connected via a network, as part of a cluster or cloud configuration.

3.1.1 Homogeneous Systems

This section presents resource sharing techniques for standard homogeneous systems where a single processor architecture is available. Prior work investigates software and architecture solutions that enable efficient system sharing and high performance.

SOS[98] is a job scheduler which combines an overhead-free sample phase that collects information about the possible schedules on the system and a symbiosis phase which uses the collected information to predict the schedule that will deliver the highest processor utilization and system performance. This scheme enables efficient scheduling for systems equipped with simultaneous multi-threading processors where multiple jobs compete for resource usage. The paper of Cazorla [17] investigates system sharing for multi-threading processors too. They investigate the unpredictable performance delivered for real time applications and they present a resource management scheme that eliminates performance unpredictability in SMTs.

The paper of Eyerman [34] proposes a probabilistic job symbiosis modeling. It predicts which sets of jobs will lead to positive or negative symbiosis when co-scheduled without requiring prior co-schedule evaluation. It does not require sampling phase or heuristics while it preserves system-level priorities. It also readjusts the job co-schedule continuously and introduces low overhead.

Time-sharing on shared memory multi-processors is investigated in the paper of Tucker [104] where the authors present a scheme where they maximize application performance while they enable fair resource sharing. The dynamic nature of multi-tasking systems is considered where the system load is continuously varying and a scheduling scheme is proposed which controls the number of active processes.

A technique for mitigating contention for shared resources via thread scheduling is provided by Zhuravlev [114]. A comprehensive analysis of contention mitigating techniques that rely on software scheduling and a thread classification scheme are presented. The proposed solution reduces contention for the cache hierarchy, memory controller, memory bus and hardware pre-fetching and also enforces quality of service.

The paper of Gabor [36] enables fairness and throughput improvements on systems with *switch on event multi-threading*. This type of multi-threading reduces the power requirements by switching threads on execution stalls. However, the original scheme does not consider fairness, a problem that is solved by this paper. This work also defines a fairness metric by using the ratio of individual thread speedups.

The paper of Ghodsi [41] investigates fairness in computer networking where multiple resources need to be shared in parallel and proposes a new algorithm, named *Dominant Resource Fair Queuing (DRFQ)*. This algorithm retains the attractive properties that fair sharing provides for one resource. It generalizes the concept of virtual time in classical fair queuing to multi-resource settings. The algorithm is generally applicable in contexts where several resource types need to be multiplexed.

The technical report of Jain [49] investigates fairness for resource sharing in various areas of computer science. It then proposes a quantitative fairness metric named *Fairness Index* that is applicable to any resource sharing or allocation problem.

FST[32], *Fairness via Source Throttling*, is a mechanism that enforces fairness in the entire memory sub-system of modern computer architectures. Its design eliminates the need for developing fairness mechanisms for each individual resource of the memory sub-system. This work provides metrics for quantifying unfairness at system level. It enforces thread priorities, supports different fairness objectives and fairness-performance trade-offs.

This section presented software and hardware solutions for resource sharing on standard homogeneous architectures. The discussed publications provide significant functionality and performance improvements for single architecture systems but they do not manage the sharing of accelerator resources found in heterogeneous systems.

3.1.2 Heterogeneous Systems

This section presents resource sharing techniques for heterogeneous systems. Here, a system typically comprises multiple processor types. Resource sharing on heterogeneous systems faces the challenges found in commodity systems and additional complexity introduced by diverse processor architectures.

Elastic Kernels[85] proposes a technique that improves graphic processor utilization and leads to increased system throughput. This work exploits concurrent kernel executions using multi-program workloads to achieve higher performance. The technique involves code transformations that enable resource allocation control.

Spatial multi-tasking on graphic processors has been proposed in paper Adriaens [1]. This work analyzes pre-existing, multi-tasking techniques such as cooperative and preemptive multi-tasking which partition GPU time among different applications and indicates weakness of these approaches. It then introduces spatial multi-tasking where GPU resources are partitioned among multiple applications simultaneously. The work

concludes by demonstrating the benefits of using spatial multi-tasking instead of, or in combination with, preemptive or cooperative multitasking.

The paper of Guevara [43] proposes the static merge of workload codes that are dispatched to GPUs for computation. This technique improves the utilization of GPU resources and delivers throughput speedups both for workload kernels that underutilize the processor resources and kernels that have a memory-bound behavior.

The multikernel [9] proposes a new Operating System architecture for multi-core heterogeneous systems. The proposed architecture aims to efficiently manage systems consisting of large number of processor cores of different architectures, memory hierarchies, interconnects, instruction sets and Input/Output configurations. This work adopts a distributed design where individual cores operate independently and exchange messages for system level coordination. Finally, it explores key Operating System operations, such as memory management, that typically require central management and demonstrates how they can be implemented on the proposed distributed scheme.

TimeGraph [56] is a real-time scheduler, developed at driver level, that enables GPU multi-tasking for real-time graphics applications. It guarantees minimum performance interference for graphic workloads by adopting a new event driven model that prioritizes computation requests made by user-space applications. It deploys a resource reservation mechanism and a predictive execution cost model in order to enforce fair GPU sharing and quality of service. TimeGraph maintains the frame-rates of primary GPU tasks at the desired level even in the face of extreme GPU workloads, whereas these tasks become nearly unresponsive without TimeGraph support.

Fair accelerator scheduling is also explored by Menychtas [75]. This work indicates the problematic management of modern accelerators performed by commodity Operating Systems and proposes explicit accelerator scheduling and resource management. It proposes a disengaged scheduling strategy in which the kernel intercedes between applications and the accelerator, monitors accelerator usage by different applications and enforces fairness by prioritizing kernel execution requests among the applications. Multiple scheduling policies are supported.

GPUfs [97] eases GPU programmability and integration by making the existing file system directly accessible from the GPU code. This work provides a POSIX-like API that accesses directly file streams from the Filesystem of the Operating System. It exploits GPU parallelism and enables GPU file access by extending the buffer cache into GPU memory. It enables extensive Input/Output operations from the kernel code and leads to performance improvements.

This section discusses solutions for sharing accelerator resources. The reviewed papers provide software solutions that require the static merge of different applications, hardware modifications or Operating System changes. This thesis considers their findings and directly compares against them when it is appropriate.

3.1.3 Inter-Node Accelerator Resource Sharing

This section reviews prior work related to inter-node sharing of accelerating resources. The reviewed solutions enable accelerator sharing across the nodes of a cluster or cloud infrastructure.

A runtime infrastructure that enables the network abstraction and sharing of GPUs is provided by Becchi [11]. It improves concurrent accelerator usage, while it applies a virtual memory abstraction via a memory manager where each application operates in isolation. This work enables multiple scheduling policies, dynamic binding of applications to GPUs, load balancing and resilience to accelerator failures. The proposed solution can be either integrated with cloud infrastructures and enable the virtualization of accelerator resources or be part of the integrated resource management infrastructure for heterogeneous clusters in High Performance Computing.

VOCL [111], which stands for virtual OpenCL, is a framework that enables transparent access and utilization of local and remote GPU resources. This work relies on the OpenCL programming model and enforces the virtualization of physical GPUs that can be transparently managed. It requires no source code modifications or changes to existing applications. This work concludes by proposing strategies to minimize the communication overhead introduced by VOCL.

Libwater[42] provides a uniform approach for programming distributed heterogeneous computing systems. It extends the OpenCL programming model by introducing an additional but simple programming interface. Libwater also consists of a runtime environment that handles OpenCL operations across multiple nodes instead of standard OpenCL that is limited to a single node. Furthermore, Libwater provides an enhanced event system that enables inter-context and inter-node device synchronization. The event system is used for the construction of Directed Acyclic Graphs of computation dependencies. A series of runtime optimizations are then built on the top of the graph.

rCUDA[30] is a framework that enables remote GPU acceleration in HPC clusters which allows a reduction in the number of accelerators required for the cluster operation. This leads to energy, maintenance, space and cost savings. It extends the CUDA

programming model in order to access and control GPUs on remote nodes while it also natively supports GPUs that are locally attached.

This section reviewed work on network sharing of accelerator resources. The considered publications combine inter-node communication protocols with heterogeneous computing frameworks and enable remote access to accelerators. This type of work tends to consider HPC application scenarios with limited multi-tasking capabilities.

3.2 System Resource Virtualization

This section reviews prior work for system resource virtualization targeting operations isolation, resource sharing control and deployment of secure computing environments. Virtualization solutions are categorized in Hypervisor-based virtualization where the Operating System performs on the top of a virtual environment and Operating-system-level (OS-level) virtualization where a virtual environment takes place for every process. Prior work on Hypervisor-based virtualization is provided in section 3.2.1. Section 3.2.2 discusses work in the area of Operating-system-level virtualization. Finally, section 3.2.3 presents existing solutions for the virtualization of accelerator resources.

3.2.1 Hypervisor based Virtualization

This section presents prior work and solutions for Hypervisor-based virtualization where Operating Systems operate on the top of virtualized hardware resources.

Xen[6] is an x86 virtual machine monitor which allows multiple commodity Operating Systems to share standard hardware in a safe and resource managed manner. Xen provides a virtual machine abstraction and requires every Operating System to have special driver support for efficient virtualization. The Xen design scales up to 100 virtual machines on a server and introduces a small overhead when compared with unvirtualized systems.

Kvm [61] is the default virtualization solution of Linux kernel. It operates as an independent Linux sub-system and leverages the virtualization extensions of modern processors to enable Operating System virtualization with negligible overhead. The Linux kernel treats the virtual machines monitored by Kvm as regular processes. The Kvm operations are fully and seamlessly integrated with the Linux kernel.

A quantitative comparison of Xen and Kvm is given by Deshane [27]. The study focuses on overall performance, performance isolation and scalability of virtual ma-

chines.

The Input/Output and multi-core scaling challenges that virtualization faces in High Performance Computing are discussed by Gavrilovska [37]. This paper analyzes and quantifies these issues and then proposes new methods for device virtualization that improves I/O performance. It also describes techniques for Quality of Service and reliability.

The approach of VMware hypervisor for the virtualization of Input/Oupput devices is presented by Sugerman [102]. This approach relies on the Operating System of the host platform in order to virtualize the CPU and memory resources. Furthermore, it uses the existing drivers and services of the host system for the Input/Output operations of virtual machines. It does that by providing virtual Input/Output devices and performing a set of optimizations that minimize the CPU utilization for this type of virtualization.

Support for soft real-time requirements in the context of virtualialized environments is proposed by Lee [66]. The authors indicate that modern hypervisors have not been designed for soft real-time applications due to the low performance of Input/Output virtualization, increased scheduling latencies and shared-cache contention. They propose a new virtual machine scheduler that manages scheduling latency as a first-class resource and performs cache management. Its revised load balancing mechanism also minimizes delays.

CloudScale [94] is a system that performs fine-grained resource sharing for multi-tenant cloud computing infrastructures and requires minimum resource provisioning. It relies on online, adaptive resource demand prediction and it does not require any prior knowledge of the application workloads. It is implemented on top of Xen.

The mapping of HPC applications to hybrid infrastructures of dedicated and cloud resources is explored by Gupta [44]. The work is logically divided in two parts. First, application characterization takes place by analyzing the application performance on both dedicated clusters and cloud. Then, an algorithm uses the characterization results and provides an efficient mapping.

This section considered state of the art solutions for hypervisor-based virtualization of systems resources. The reviewed publications consider homogeneous systems and emphasize on lightweight virtualization for Operating Systems.

3.2.2 OS-level Virtualization

This section presets virtualization solutions for the isolation of individual system processes. This type of virtualization enforces both secure execution environments and resource allocation control.

The paper of Banga [5] presents and evaluates a new operating system abstraction, named *resource container*, which separates the process protection mechanisms from the system resource management. The new abstraction enables fine-grained resource management across system processes and enables the development of robust servers. The proposed design exposes a simple and firm control over priority policies.

Virtualization use cases, such as HPC, that require both a high degree of isolation and efficiency are considered by Soltesz [99]. This work presents an alternative to hypervisor virtualization that delivers better results for the considered application areas. The proposed solution provides an Operating System abstraction of resource and security containers. The container mechanisms are applied to general-purpose, time-shared Operating Systems. This work specifically describes the design and implementation of Linux-VServer which leads to higher performance than Hypervisor solutions.

PDS[2] is a virtual execution environment that enables efficient software development and central management. It enables a partial system virtualization and supports the reproduction of virtual environments across different host machines. A framework permits optimizations that require semantic awareness that is not available at the Operating system level.

Denali[108] is an isolation Operating System kernel designed to safely multiplex large numbers of untrusted processes on shared systems. It achieves this by defining a virtualization abstraction at the Operating System level where it enforces security and resource management. This type of resource sharing reduces the cost and effort of managing physical systems.

Tessellation [21] is an Operating System design targeting Quality of Service for diverse workloads including high-throughput parallel, real-time, and interactive applications. Tessellation distributes system resources to groups, named *cells*. It then performs a two level management of system resources. The first level allocates resources at cell level, while the second manages resources within the cell by serving specific application requirements.

The overhead and performance effects of dynamic resource management of cloud infrastructures are investigated by Wang [107]. This work considers capacity over-

heads and actuation delays that may occur due to frequent re-scheduling. This work quanitifies the related overheads and compares the performance delivered by hypervisor solutions agaist OS-level virtualization approaches.

Shuttle [93] is a communication mechanism that is part of the Operating System and enables the efficient communication of applications that operate in different Virtual Machines. This work considers the inter-application communication needs for enterprise-class servers, HPC clusters and fault tolerant systems. It then proposes a scheme for inter-application interaction that integrates with the existing mechanisms of OS-level virtualization and does not compromise system security or application isolation.

Docker [28] and OpenVZ [84] are production technologies that provide lightweight, OS-level virtualization. They both rely on the mechanism of Linux kernel containers.

The paper of Kim [60] presents a virtualization framework that combines multiple GPUs and treats them as a single compute accelerator. It enables the transparent adaptivity of applications written for a single GPU to multi-GPU systems, where they exploit the computational and memory resources of all the available GPUs. A runtime environment maintains a virtual, unified, device memory that is mapped to the physical memories of the individual GPUs. The OpenCL environment treats this virtual memory as if it were the memory of a single GPU. The framework automatically distributes at run-time OpenCL kernels written for a single GPU to multiple kernels that perform in parallel on different GPUs. It also applies a run-time memory access range analysis to kernels by performing a sampling run and it identifies the optimal kernel distribution.

This section reviewed publications on OS-level virtualization which supports isolation and resource allocation control for individual processes. This type of virtualization, which is extremely lightweight, is highly relevant to the virtualization of heterogeneous resources which is presented in this thesis.

3.2.3 Accelerator Virtualization

This section presents virtualization solutions for accelerator resources.

The techniques and system architecture for the virtualization of GPU resources on VMware products are provided by Dowty [29]. This work provides a full design that takes advantage of hardware acceleration and delivers performance results comparable to native GPU usage. However, the various graphics stack implementations and

applications lead to distinct performance variations.

gVirt[103] is a product level GPU virtualizaton infrastructure that enables full GPU virtualization by running native drivers as part of the guest Operating System. This virtualization approach exploits mediated pass-through which delivers high performance, scalability and secure isolation. gVirt permits Virtual Machines to directly access GPU resources without requiring hypervisor intervention.

The work of Yang [112] uses the PCI pass-through technology in order to make GPU accelerators available to Virtual Machines. This infrastructure enables Virtual Machines to directly access GPU resources and perform computations on them via the CUDA programming interface. An evaluation is given where the proposed infrastructure is compared against native GPU usage and open source virtualization solutions.

vCUDA [95] is a virtualization solution for GPU computing. It allows applications running on Virtual Machines to directly perform computations on GPUs via the CUDA API. vCUDA intercepts and redirects CUDA calls and uses an efficient Remote Process Call scheme. This scheme forwards computation requests to the hypervisor which performs the actual calls to the CUDA driver.

Gdev [57] enables resource management for GPUs as part of the Operating System. Its functionality enables efficient GPU sharing and accessibility both for user-space applications and Operating System operations. It provides a virtual memory manager for GPUs that enables data swapping for excessive memory resource demands and efficient data sharing. Furthermore, Gdev virtualizes physical GPUs and provides multiple logical instances. This way it enforces process isolation and secure multi-tasking. Gdev authors have ported filesystem operations to take advantage of GPU resources and report significant performance improvements.

This section reviewed accelerator virtualization techniques for hypervisor systems. The presented approaches are typically vendor specific and they suit HPC scenarios instead of multi-tasking use cases due to the high overhead they introduce.

3.3 Workload Scheduling on Heterogeneous Systems

This section discusses prior work on workload scheduling for heterogeneous systems. Section 3.3.1 reviews publications on software techniques and system stack designs for workload scheduling. Prior work in computer architecture for improved accelerator scheduling is provided in section 3.3.2.

3.3.1 Runtime and Compiler Approaches

This section presents software techniques for workload scheduling on heterogeneous systems. The presented work focuses on improving performance, enhancing programmability and accelerator utilization.

SKMD [65] is a runtime infrastructure that transparently manages collaborative execution of single data-parallel kernels across multiple and asymmetric processors of different types, such as CPUs and GPUs. The developer provides a single OpenCL kernel and the required input data, SKMD then partitions the workload across the available processors and performs the computation, it then merges the partial outputs together. SKMD is also equipped with profitability heuristics for workload offloading with respect to data transfer overheads. SKMD improves the kernel execution performance by leveraging all the processor resources in parallel.

Concord [7] is a C++ compiler framework that enables GPU acceleration of a wide range of applications with minimal changes. Concord is accompanied by a low-cost runtime environment that provides Shared Virtual Memory between CPU and GPU cores which enables seamless sharing of pointer-containing data structures. This way software running on CPU and GPU can share complex data structures without code changes or explicit data transfers. Concord's compiler also performs GPU specific optimizations. This work delivers performance improvements and energy efficiency.

Lime [31] is a Java-compatible language targeting heterogeneous systems. The language type system and annotations allow the compiler to generate and optimize high quality GPU code. The high level language semantics enforce isolation and immutability invariants. These semantics provide the necessary guarantees for advanced compiler optimizations and efficient and transparent management of data transfers.

ADSM[39], Asymmetric Distributed Shared Memory, is a programming model that maintains a logical memory space for CPUs to access objects in the accelerator physical memory. This asymmetric design supports lightweight runtime implementations. The proposed model requires developers to assign data objects to performance critical code. If a critical code is selected for accelerator execution, its associated data objects are allocated in the accelerator memory. However, ADSM makes these allocations visible to the host code via the shared logical memory space. This model simplifies accelerator programmability.

The paper of Udupa [105] provides the design and implementation of a runtime and compiler infrastructure for the execution of *StreamIt*, a programming language for

streaming applications, on GPUs. A program written in StreamIt consists a graph that represents task, data and pipeline parallelism. This work presents an efficient mapping of this graph to GPU resources and proposes a software pipeline technique for efficient execution that relies on solving an integer linear program (ILP). This work also proposes data layout transformations for improving the performance of GPU executions.

The work of Shou [96] maps OpenMP annotated software to GPUs and provides *DSM*, a software distributed shared memory system that establishes a logic shared memory space and transparently manages data communication between CPUs and GPUs. This work optimizes DSM operations with a compiler assisted data prefetching scheme.

The paper of Phothilimthana [87] presents an auto-tuning framework for performance portability across various heterogeneous systems equipped with different processor types. The programming systems and memory models differ dramatically making performance portability a significant problem. This work presents a empirical model that takes place at application installation time and delivers an improved mapping of programs to processors and memories. Programs provide descriptions in how their individual algorithms may work and a compiler generates multiple versions of the code that perform different mapping of workloads across the available processors. The empirical model evaluates the available versions at installation time and provides an efficient mapping.

Yang [113] presents an optimizing compiler for general purpose computing on GPUs. It focuses on the effective utilization of GPU memory hierarchy and parallelism management. In addition to standard compiler analyses, it detects memory access patterns and performs automatic vectorization, memory coalescing, tiling and unrolling. It also performs thread block remapping or address-offset insertion for partition-camping elimination.

This section reviews prior work on runtime and compiler techniques for improved programmability and performance optimizations on heterogeneous systems. The reviewed publications provide significant performance improvements but they only consider single application performance and static workload configurations.

3.3.2 Computer Architecture

This section presents prior work in computer architecture for improved accelerator scheduling which delivers performance improvements and better accelerator utiliza-

tion.

Fung [35] considers performance degradations caused by diverging control flow executions on GPUs. The authors solve this problem by proposing a stack scheme that allows different SIMD processing elements to execute diverging program paths after branch instructions. The proposed technique dynamically regroups threads into new warps based on their control flow behavior.

Narasiman [79] proposes GPU architecture changes in order to improve resource utilization. Standard GPU architectures group threads into warps and thread blocks. This work extends this scheme by proposing a large warp concept and two-level warp scheduling. This way it reduces stalls due to long latency operations, such as memory accesses and conditional branch instructions.

Chen [20] investigates thread scheduling algorithms for GPUs and indicates that classical round-robin schemes are inefficient in scheduling instructions and memory accesses with disparate latencies. It then proposes a flexible scheduling policy that supports flexible round-robin distance for efficient utilization of multi-thread parallelism. This scheme also allows more overlaps between short-latency compute instructions and long-latency memory accesses.

CCWS [89] is a thread scheduling scheme for GPUs that is cache aware. It is equipped with an intra-wave locality detector for memory accesses. CCWS shapes memory access patterns to avoid thrashing the shared L1 caches. This way it outperforms replacement-based scheduling schemes.

Huo [45] presents a scheduling mechanism to efficiently support recursion in modern GPUs. Code recursion causes diverging control flow for the various threads of a kernel execution and leads to significant performance slowdowns. This work provides a set of scheduling policies that solve this problem.

OWL [53] is a thread block aware scheduling mechanism for GPUs. It delivers improved performance by tolerating long memory latencies. The mechanism supports four schemes in order to minimize the impact of long latencies. The first two schemes, *CTA-aware two-level warp scheduling* and *locality aware warp scheduling* reduce cache contention and improve the latency hiding. The third scheme, named *bank-level parallelism aware warp scheduling*, improves performance by enhancing DRAM bank-level parallelism. The last scheme performs memory-side prefetching to enhance performance by taking advantage of open DRAM rows.

Brunie [16] presents two techniques that mitigate the impact of thread divergence on GPUs. The proposed techniques relax the Single-Instruction Multiple-Thread (SIMT)

model by allowing two distinct instructions to be scheduled to disjoint subsets of the same row of execution units, instead of one single instruction. This relaxed model enables more thread grouping opportunities and improves GPU utilization.

This section reviewed prior work on computer architecture techniques that optimize the performance of workload executions on accelerators. The presented work requires significant architecture changes which are not available today and emphasizes on single workload executions instead of multi-tasking scenarios. The contributions of this thesis are orthogonal to the reviewed architecture work.

3.4 Memory Management and Data Sharing

This section discusses prior work on memory management, data transfers and data sharing techniques. Section 3.4.1 discusses memory allocation on homogeneous and heterogeneous systems. Resource management for Non-Uniform Memory Access Architectures (NUMA) is discussed in section 3.4.2. Finally, section 3.4.3 presents communication optimizations for heterogeneous systems.

3.4.1 Memory Allocators

This section discusses prior work on memory management for parallel systems conducting multi-tasking computations.

Hoard [12] is a scalable memory allocator that delivers high performance with efficient memory management for parallel applications such as web servers, database managers and scientific applications. It efficiently tackles problems found in traditional memory allocators, such as poor scalability and performance, and heap organizations that introduce false sharing. Hoard leverages a global heap and per-processor heaps in a scheme that minimizes synchronization overheads and dramatically reduces memory fragmentation.

Heap layers[14] is a C++ template framework library that provides a powerful infrastructure for building custom and general purpose memory allocators that deliver high performance. Developers are given building blocks via a clean, easy-to-use allocator interface that allows fast and error free design of general purpose memory allocators that compete the performance of state of the art solutions, or specialized allocations for particular application needs.

TCMalloc[40] is a user-space memory management library developed by Google.

It delivers higher performance than standard memory management solutions found in production systems. It achieves that by supporting low overhead memory management operations that scale to parallel multi-core systems. TCMalloc reduces lock contention, where there is virtually zero contention for small objects. If lock operations are required, spinlocks are used which introduce lower overheads than traditional mutual exclusion mechanisms such as mutexes. The design of the user-space memory manager of the FreeBSD Unix project is given by Kamp [55].

SFMalloc [92] is a dynamic memory allocator for multi-threaded software. Each thread manages its own local heap and local memory management does not require any synchronization. SFMalloc design rarely uses synchronization and in cases where it is required lock free mechanisms are used. It also exploits memory block caching mechanisms in order to reduce the overhead of memory management operations. Freed blocks or intermediate memory chunks are cached hierarchically in each thread's heap and they are used for future memory allocation.

DieHard [13] is memory management runtime that makes software tolerant to memory errors such as buffer overflows, dangling pointers and reads of uninitialized memory. DieHard approximates an infinite-sized heap by using randomization and replication in order to probabilistically achieve memory safety. DieHard is equipped with a memory manager that randomizes the location of objects in memory. In addition, DieHard supports a replicated mode, where multiple instances of the same application run in parallel and agreement on the output is required. DieHard's resilience in memory errors can reduce program crashes, security vulnerabilities and unpredictable behavior without requiring changes in the original application code.

RSMV [51], which stands for Region-based Software Virtual Memory (RSVM), is a virtual memory mechanism that resides on both CPU and GPU in a distributed and cooperative manner. It enables automatic GPU memory management and transparent data transfers between CPU and GPU memories. It enables kernel-issued on demand data fetching from the host into the GPU memory. It also permits transparent GPU memory swapping into the main memory and scales GPU kernels to large problem sizes that originally cannot fit into the GPU memory.

This section reviewed prior work on memory management for parallel and heterogeneous systems. This thesis work requires advanced memory management and relies on the findings of the reviewed publications.

3.4.2 Non-Uniform Memory Access Architectures

This section reviews prior work on resource management for Non-Uniform Memory Access (NUMA) architectures. These systems have multiple CPU sockets with local RAM interfaces that are called NUMA nodes. Intra-node local memory accesses typically are faster than accesses to remote nodes' memories.

A NUMA aware version of Google's TCMalloc memory manager is presented by Kaminski [54]. This work makes the memory manager aware of NUMA topologies, the number of nodes and the local memory available to each of them. The memory allocation approach has been modified to consider memory location and the applied practices attempt to minimize the inter-node communication overhead.

Larowe [62] proposes an Operating System design that efficiently manages data placement and communication on complex NUMA architectures. The memory management subsystem has been modified to support a modular interface for multiple memory management policies which improve system performance for existing applications without requiring any changes in the programming model.

Carrefour [25] is a memory placement algorithm for NUMA architectures that minimizes the congestion on memory controllers and interconnects, caused by memory traffic of data-intensive applications. Modern NUMA hardware delivers small inter-node communication overhead and the cost of remote memory accesses which is solved via access locality optimizations is not the main performance concern. Traffic congestion is the main cause of performance slowdowns and Carrefour solves this.

This section reviewed publications for memory management and data placement on Non Uniform Memory Access (NUMA) systems. The thesis contributions consider NUMA architectures and use the findings of the reviewed papers for memory management and data placement.

3.4.3 Communication Optimizations for Heterogeneous Systems

This section reviews communication optimizations for heterogeneous systems. These optimizations typically target the reduction of communication overhead either by improving performance or eliminating redundant data transfers.

Jablin [48] presents a fully automatic system for managing and optimizing CPU-GPU communications. The proposed system combines a run-time environment and a set of compiler transformations while its operation does not rely on static code analysis or programmer given annotations. This work simplifies manual GPU par-

allelization while it improves the applicability and performance of automatic GPU parallelizations. The proposed scheme suffers from inefficiencies of alias analysis and type-inference. DyManD[47] overcomes these limitations and enables automatic management for complex and recursive data-structures without any need for static analysis. This is done via memory alias checks and type-inference at run-time.

Lee [64] investigates data streaming and data compression in order to reduce the communication overhead introduced by heterogeneous computing. The data streaming technique enables overlap of communication and computation, while data compression reduces the data sizes transferred between the host and GPU memories. Both techniques introduce additional overhead but their data manipulation leads to significant reduction of communication overheads and deliver overall performance improvements. Two case studies of radix sort and 3-start are discussed.

MVAPICH2 library [106] is a novel MPI design that unifies the management of CUDA and MPI data transfers and exposes a single MPI interface. This interface enables direct communication with GPUs for both read and write operations. Data transfers from GPU and network are now overlapped. MVAPICH2 reduces one way communication latencies and increases the throughput of collective operations.

White III [109] explores performance improvements from overlapping computation and communication operations on hybrid clusters equipped with GPUs of various generations and different types of interconnect. The evaluation considers computational software written in Fortran, MPI, OpenMP and CUDA. The authors find that overlapping CPU computation, GPU computation, parallel communication, and CPU-GPU communication can provide performance improvements of more than a factor of two.

A GPU aware implementation of the MPI runtime is given by Ji [50]. Standard MPI implementations are unaware of accelerators and their memory hierarchy. This, typically, leads to additional copy operations which introduce significant overhead. The proposed runtime is aware of GPU resources and leads to efficient intra-node communications across GPUs and eliminates redundant copies.

CudaDMA[8] proposes architecture and programming model changes targeting improved data transfers between the on-chip and off-chip memories of GPUs. DMA warps improve memory bandwidth utilization by better exploiting the available memory-level parallelism and by introducing inter-warp producer-consumer synchronization mechanisms. DMA warps simplify the CUDA programming by decoupling the need for thread array shapes to match data layout.

CUBA[38] is an architecture where accelerators have direct access to the required input and output data without the need for explicit data transfers. This is achieved by mapping data structures required for kernel computations between the host and accelerator memories. The proposed mapping, which does not require any data layout transformation, minimizes and optimizes data transfers between the host and accelerator memories. This approach reduces communication overheads and simplifies the programming model because there is no need for explicit data transfer management. The mapping is built on the top of a cache mechanism that has a selective write-through policy.

Dymaxion[19] is an API extension for CUDA that enables memory mapping optimizations which improve the efficiency of memory accesses on GPUs. Dymaxion requires the developer to specify memory layout remapping and index transformation functions. These functions are then used by a runtime library that transparently transforms the data layout of input and output buffers to representations that lead to improved memory access performance on GPUs. The data layout transformation overlaps with host-device communication, this way the overhead remains low.

This section presents prior work on communication optimization for heterogeneous systems. The reviewed papers provide programming and architecture techniques that optimize data communication on heterogeneous systems. One of they key contributions of this thesis is a communication optimization which is orthogonal to the presented papers.

3.5 Performance Evaluation & Modeling on Accelerators

This section reviews prior work on techniques and frameworks for the performance evaluation and modeling of heterogeneous systems. It also reviews benchmarks suites for heterogeneous computing.

Coutinho [23] presents a profiling tool that relies on performance predicates. The tool quantifies the major sources of performance degradation. It uses the collected information to provide hints to the developer on how to improve his code.

TAUcuda[68] package is a performance measurement technology for CUDA applications that integrates with the TAU parallel performance system. The design of this package relies on experimental software stack and provides detailed performance

information for kernel executions and CUDA operations. Furthermore, this work does not require any modification of the program source or executable code.

Baghsorkhi [4] presents an analytical model for predicting the performance of computations on GPUs. The model analyzes GPU kernels by identifying how they exercise the GPU hardware features. Each kernel has a work flow graph representation which is used for execution time estimation. The model is either used by an auto-tuning compiler for selecting the more promising optimizations or can be used by application developers as a profiling tool. The proposed model captures full system complexity in detail and shows high accuracy in predicting the performance trends of different kernel optimizations.

Keeneland[100] is a performance evaluation system for NUMA systems equipped with multiple GPUs. It keeps track of complex performance phenomena and detects contention for shared system resources. This work concludes its contributions by proposing programming strategies that maximize performance and system utilization.

McCurdy [73] investigates the performance behavior of multi-threaded scientific software on NUMA systems. It describes the performance problems that NUMA systems may have for parallel software and demonstrates how modern NUMA system behavior may lead to scaling failures. The authors propose methods which use hardware performance counters and detect performance bottlenecks. Mephis is a proposed toolset that relies on Instruction Based Sampling and pinpoints inefficient memory accesses.

Parboil[101] benchmark suite is a set of throughput computing applications which can be used for the performance evaluation of accelerator architectures, runtimes and compilers. The suite provides multiple implementations where different programming models and paradigms have been used. It also provides benchmark versions of varying levels of optimization.

Rodinia[18] is a benchmark suite for heterogeneous computing. It provides applications and kernels which target commodity CPU cores and GPU platforms. Furthermore, this work provides a characterization of the benchmarks. The selected benchmarks cover a large number of parallel communication patterns, synchronization techniques and power needs.

This section reviews prior work on performance evaluation, performance modeling and benchmark suites for heterogeneous systems. This thesis presents extensive performance evaluations on heterogeneous systems which rely on this prior work.

3.6 Summary

This chapter has provided, to the best of the authors knowledge, a brief review of prior work on the various areas touched upon in this thesis. It has covered resource sharing and virtualization techniques for homogeneous and heterogeneous systems. Furthermore, workload scheduling on accelerators and various methods for optimizing data communication for software using accelerators have been discussed. Methods for memory management, and resource management techniques on multi-node shared memory systems have also been presented. Finally, this chapter reviewed performance evaluation and modeling techniques.

The next chapter presents an approach that reduces host-device communication overhead for OpenCL applications. It does this without requiring modification or re-compilation of the application source code and is portable across platforms.

Chapter 4

Host-Device Communication Optimization

This chapter develops an approach that reduces host-device communication overhead for OpenCL applications. It does this without modification or recompilation of the application source code and is portable across platforms. It achieves this by tracing and analyzing calls to the runtime made by the application and then selecting the best platform specific memory allocation and communication policy. This approach was applied to 12 existing OpenCL benchmarks from Parboil and Rodinia suites on 3 different platforms where it gives on average a speedup of 1.51, 1.31 and 1.48, respectively. In certain cases, our approach leads up to a factor of three times improvement over current approaches.

The remainder of the chapter is organized as follows. An introduction of the challenges and contributions of this chapter are given in section 4.1. Section 4.2 describes the motivation for our work. Section 4.3 presents a high level overview of our host-device communication optimization. Section 4.4 describes our approach for dynamic platform characterization. The application tracing and analysis that support the optimization are discussed in sections 4.5 and 4.6, respectively. The runtime environment that performs the optimization is discussed in section 4.7. Sections 4.8 and 4.9 present the experimental setup and the evaluation results. Section 4.10 provides a summary of this work.

4.1 Introduction

Modern computer architecture design has shifted from single core to parallel heterogeneous systems. Here, the OS and the legacy application stack typically run on a number of commodity cores while GPU accelerators are exploited through workload dispatch performed by the application. OpenCL[58] and CUDA[81] are popular frameworks providing workload dispatch and data transfer management between the distinct memory sub-systems of the host and GPUs.

Although OpenCL and CUDA provide a unified interface for utilizing GPUs, trying to optimize applications and estimating their performance remains a challenging task. This is largely due to a lack of transparency. GPU architectures and the data exchange protocols between the distinct memories are opaque and differ in behavior that can lead to performance loss.

Currently, developers have to spend time tuning their code to each new GPU based platform. Given the rate of GPU hardware evolution, this approach is time consuming and is a significant barrier to heterogeneous computing. So while we have portable code, we do not have portable performance.

Our approach to address this problem is to develop a portable framework targeted at the optimization of host-device OpenCL communication. As we show, such communication can be a significant overhead in OpenCL applications. By improving data transfers between the different memory sub-systems, we can achieve significant performance improvements automatically without altering the user's application code.

Programming guidelines from GPU vendors recommend the use of memory locked segments (memory pages are pinned in main memory) for the maximum exploit of interconnect bandwidth [81]. Such an approach leads to improved communication performance but with the penalty of prohibitive allocation times. It violates application portability, as multiple assumptions about the hardware and OS are hard-wired into the code. Current approaches therefore trade-off portability for performance.

Our work solves this problem by redirecting dynamic memory allocation requests, selecting the memory allocation policy that best improves host-device communication and reduces overall application time. Our host-device communication optimization remains transparent to the application execution and does not require any alteration or re-compilation of the application. To achieve this, we need knowledge of the particular allocation and communication policies of the target platform and their associated costs. This is achieved by an automatic platform characterization scheme.

As well as platform characterization, our approach requires knowledge of application behaviour. Specialized memory allocation is only of benefit, if the memory allocated is involved in host to device communication. Using specialized memory allocation for data that is used solely on the host incurs excessive overhead. To determine if a memory allocation requires special treatment, the application is traced. We record calls to OpenCL and memory allocation functions as well as tracking data dependencies in SSA form. The traces are condensed into a compressed call trace so to enable later analysis to determine candidate memory allocations for optimization. After analyzing the compressed trace, we can determine those memory allocations that are involved in host-device communication. This is used to determine the best memory allocation and communication policy for each call site. This information is then used at runtime to redirect dynamic requests appropriately. This is all performed transparently to the user.

The chapter contributions are summarized as:

- A platform characterization scheme that automatically detects the host-device communication and memory allocation capabilities of a platform.
- An on-line tool that traces heterogeneous applications and stores in a compressed call trace.
- An optimization that analyses the compressed call trace to determine the best allocation and communication policy per call.
- An empirical evaluation that shows significant improvement over 3 platforms without any alteration or recompilation of the application program.

4.2 Motivation

This section presents a motivating example illustrating the communication optimization enabled by our framework. Data is initially allocated on the host, typically via the default memory allocator via a call to *malloc*. As this data is to be used by the device, a data transfer is made. Once the data transfer is complete, kernel execution on the GPU can start. After the kernel completes, data may be transfered back to the host memory.

When standard memory allocation is used, the device driver is forced to transfer data page by page with multiple DMA transfers from host to device as shown in the

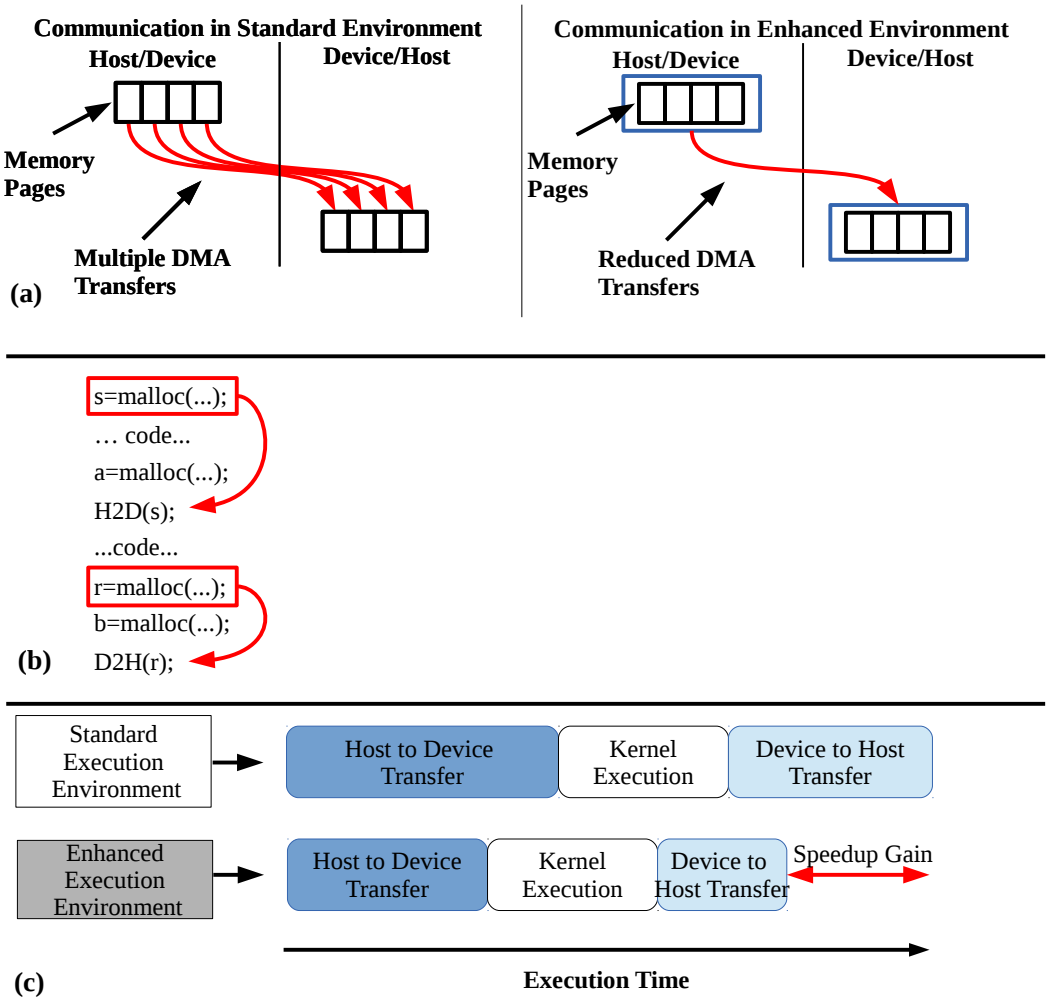


Figure 4.1: Our optimization reduces the number of DMA transfers for host-device communication by using specialized memory allocation which guarantees memory pages are in physical memory (Figure a). We use compiler techniques to determine the memory allocations that will be served by our optimization with specialized memory allocation (Figure b). This improves the overall execution time of an application by reducing the host-device communication overhead (Figure c).

left side of figure 4.1a. This is because it needs to handle page fault for pages that are not present in main memory but stored on disk.

In our approach we exploit special memory allocation policies that provide memory locked segments. The memory pages remain permanently in main memory and are not swapped to secondary storage. The driver now has the guarantee that all pages are in main memory and performs the host to device communication with the fewest possible DMA transfers. The reduction of DMA transfers and the absence of page faults increase the interconnect utilization. This is shown in the right side of figure 4.1a.

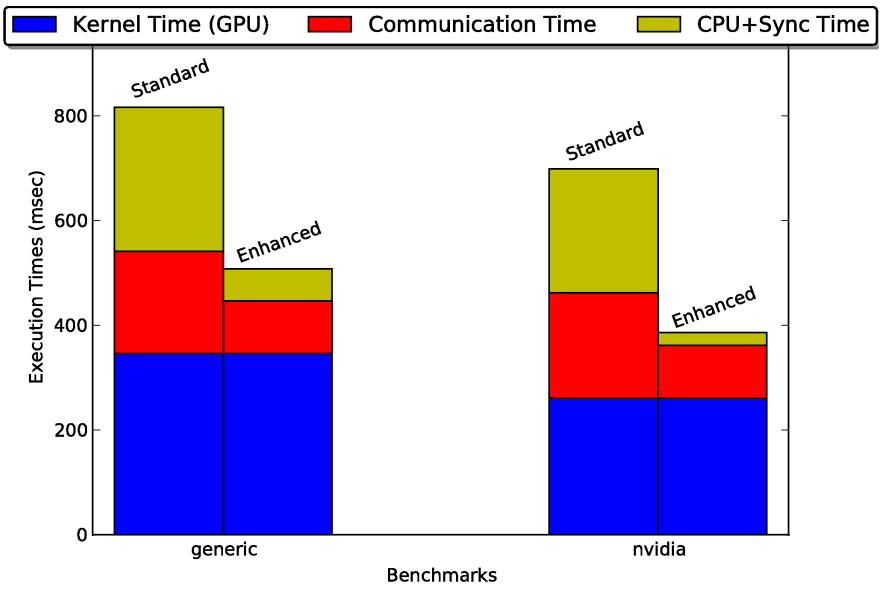


Figure 4.2: Execution time breakdowns for two versions of `mri-mridding` benchmark: `generic` and `nvidia`. The left hand bar, `Standard`, shows its execution time breakdown in the standard execution environment. The right hand bar, `Enhanced` shows the execution time with our optimization in the enhanced execution environment. The overall execution time is broken down into kernel, communication and cpu+sync subcomponents. In both cases our enhanced environment dramatically reduces the cost of communication and sync giving significantly improved execution time.

This communication optimization is only of benefit if the memory allocated is involved in host to device communication. Using specialized memory allocation for data that is used solely on the host incurs excessive overhead and massive use of limited system resources. Our approach uses compiler analysis of compressed call traces to determine those memory allocations that are associated with host to device communication. This is illustrated in figure 4.1b. Here there are multiple memory allocation calls, but only 2 of them are involved in allocating data that will take part in host-device communication. By using compiler analysis we can detect such memory allocations and change their memory allocation policy. This analysis is critical to any performance gains. Applying this technique reduces the communication overhead and hence application execution time as shown in the bottom part of figure 4.1c.

4.2.1 Performance Impact

Figure 4.2 shows the speedup delivered by our optimization to the `mri-gridding` benchmark of the Parboil suite. To illustrate the applicability of our approach, we consider two versions of the benchmark. The first labelled `generic` is the generic OpenCL implementation of `mri-gridding`. The second labelled `nvidia` is a version specially developed for NVIDIA GPUs. Both versions are available in the Parboil suite. Two bars are shown for each version. The left hand bar in each pair shows the time spent in the standard environment (`Standard`), the right hand bar shows the time spent with the optimization in our enhanced environment (`Enhanced`). In addition, each bar shows a stack which gives the total time spent executing the program broken down into subcomponents: `kernel` execution time, `communication` time between host and device and `CPU + Sync` time.

For the `generic` version, we are able to reduce the total execution time from 800ms to 500ms giving a speedup of 1.6x. This is due to the reduction of communication and cpu+sync time from 450ms to 150ms. In the case of the `nvidia` optimized version, the kernel is notably faster than the generic one, 230ms vs 280ms. However, improvement in the kernel execution time means that communication time is actually a greater percentage of application execution time. Applying our optimization technique again reduces the cost of data transfers leading to a speedup of 1.8x. As programmers tune their application kernels, the relative cost of communication grows, making such optimizations increasingly important.

4.2.2 Summary

In this chapter, we present a host-device communication optimization which is portable and transparent and does not require access or recompilation of the application code. We achieve this by characterizing the memory allocation and communication capabilities of a platform through micro-benchmarking and by performing application tracing. We apply off-line compiler analysis of the collected compressed trace to determine which allocations are associated with host-device communication . At run-time we use platform information and application analysis to redirect memory allocation and communication calls to the most profitable policy.

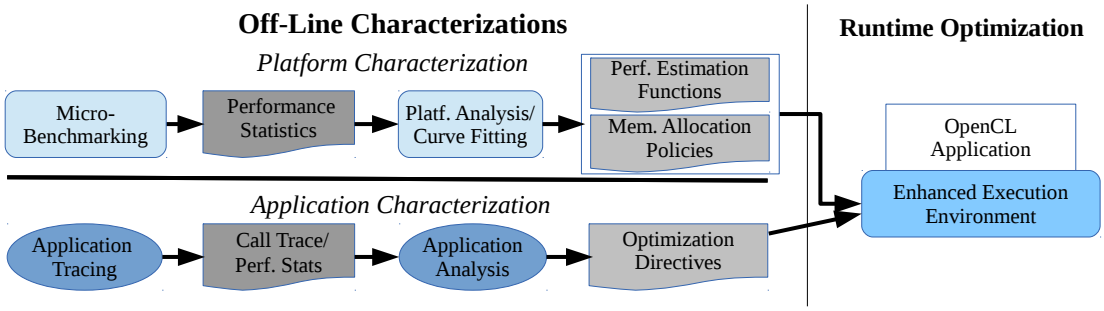


Figure 4.3: An Overview of the Host-Device Communication Optimization. Our optimization relies on the platform and application characterizations. The platform characterization detects the memory allocation and host-device communication capabilities of the platform. The application characterization traces and analyzes the application for the detection of memory allocations that are used in host-device communication. The enhanced environment uses the platform characterization and the results of application analysis for the run-time selection of memory policy that gives the highest communication speedup.

4.3 Optimization Overview

In this section, we provide a high level overview of our host-device communication optimization and the key components that support it. The optimization exploits architectural and driver features but in a portable and transparent manner with no code modification or recompilation.

The optimization is based on the observation that different memory allocation policies affect the performance of OpenCL data transfers. One example of this case is the use of programmer hard-coded memory locked segments (memory pages pinned in main memory) in applications running on platforms with GPUs, where memory locking improves the performance of OpenCL data transfers. Our work generalizes this to a wider range of memory allocation policies without any inherently unportable programmer hard-coding. Furthermore, the memory allocation and communication capabilities of the platform are detected automatically. During application execution, dynamic memory allocations are served by the policy that leads to the highest host-device communication rates on that platform.

Our optimization, as can be seen in Figure 4.3, relies on two characterizations. The first, platform characterization, is concerned with detecting the memory allocation cost and host-device communication performance of the platform. The second, application

characterization, is concerned with the application tracing and the off-line analysis of the traced data, both are required for the identification of memory allocations that should be redirected by the optimization. Combining both components allows the runtime optimization of the application. The following sub-sections provide further details of the components and the runtime optimization.

4.3.1 Platform Characterization

Here, we evaluate the allocation and communication capabilities of the target platform by using two micro-benchmarks. The first benchmark is used to measure the allocation time for different data sizes. The second is used to measure host-device communication rates for each policy. Curve fitting is applied to the data resulting in simple performance estimation functions.

This benchmarking is performed just once per platform. The allocation policies and their performance estimation functions are later used at runtime to optimize a target application. Section 4.4 describes the platform characterization in further detail.

4.3.2 Application Characterization

We trace an application's interaction with the OpenCL environment. We then analyze the resulting trace to determine the best allocation policies for later runs. Tracing is performed by monitoring application calls to the OpenCL library and memory allocation functions. Later, application analysis operates on the traced data and detects the dynamic memory allocations that are used in host-device communication operations. The detected allocations are later used as application dependent input by the runtime optimization.

4.3.2.1 Application Tracing

The application executes with a sample input while its operation is monitored by our tool. The framework monitors every application call to OpenCL and memory allocation functions, constructs a compressed call trace, builds data dependencies and collects performance statistics. The application tracing is essential for the collection of data required by the application analysis. The application tracing is further described in Section 4.5.

4.3.2.2 Application Analysis

Our optimization requires the detection of dynamic memory allocations that are used in host-device communication operations. We develop an algorithm that investigates if applying the optimization is worthwhile for the application based on a simple eligibility heuristic. If the application is optimization eligible, the algorithm detects the allocations that are used in host-device communication operations. For further details see 4.6.

4.3.3 Runtime Optimization

Our enhanced environment redirects only the memory allocation operations for the host memory that is used in host-device communication. It replaces the standard allocation policy with the one that is likely to lead to the highest bandwidth rates for host-device communication. An allocation manager operates on top of the available policies and uses the application analysis results and the performance estimation functions for the redirection decisions. The manager leverages user-space memory allocators for policies that have prohibitive allocation times. The features of the enhanced environment and the memory allocator are described in Section 4.7.

4.4 Platform Characterization

This section describes the platform characterization, where we detect the dynamic memory allocation and host-device communication capabilities of a platform through micro-benchmarking. We consider four allocation policies and for each we collect data in allocation and host-device communication performance. Curve fitting is performed on the collected data to give simple performance estimation functions.

4.4.1 Memory Allocation Policies

A memory allocation policy is considered as a set of memory management operations that are used for the dynamic allocation of a segment. Our platform characterization considers four allocation policies, the `Standard`, `OpenCL`, `Standard with Locking` and `Hybrid` policies.

Standard: This is the default memory allocator, accessible through the standard memory management functions of C/C++, e.g. *malloc*. In algorithm 4.1, we describe the remaining policies.

OpenCL: Lines 3 to 9 describe the policy where the OpenCL runtime allocates a memory segment on the host and attaches it to the application address space.

Standard with Locking: This policy is described on lines 13 to 18, where the standard allocator provides a memory segment and the POSIX specified *mlock* function locks the segment on main memory.

Hybrid: This policy is shown on lines 22 to 30 where a memory segment is allocated through the standard allocator, after it is locked in main memory through mlock and later an OpenCL buffer that exploits the segment space is created. In the end, the buffer is attached to the application address space.

```
1  /*OpenCL*/
2
3  void *oclmalloc(size_t size)
4  {
5   cb=clCreateBuffer(c, CL_MEM_READ_WRITE |
6   CL_MEM_ALLOC_HOST_PTR, size, NULL, &rv);
7   return clEnqueueMapBuffer(q, cb, CL_TRUE,
8    CL_MAP_READ, 0, size, 0, NULL, NULL, &rv);
9  }
10
11 /*Standard with Locking*/
12
13 void *lockmalloc(size_t size)
14 {
15  p=malloc(size);
16  mlock(p,size);
17  return p;
18 }
19
20 /* Hybrid*/
21
22 void *hybridmalloc(size_t size)
23 {
24  p=malloc(size);
25  mlock(p,size);
26  cb=clCreateBuffer(c, CL_MEM_READ_WRITE |
27  CL_MEM_USE_HOST_PTR, size, p, &rv);
28  return clEnqueueMapBuffer(q, cb, CL_TRUE,
29   CL_MAP_READ, 0, size, 0, NULL, NULL, &rv);
30 }
```

Algorithm 4.1: Dynamic Memory Allocation Policies. The figure shows malloc equivalent concepts for policies that support dynamic memory allocation in a way different to the default system allocator.

Both `OpenCL` and `Standard with Locking` policies are expected to provide memory locked segments with the difference that the first provides the segment through the OpenCL environment and the second through the standard system facilities. The `Hybrid` policy overcomes the lack of full cooperation between some OpenCL implementations and the standard system. Some OpenCL implementations cannot detect memory locked segments that are not allocated through `OpenCL`. The benefit of using `Hybrid` instead of `OpenCL` policy is that `Hybrid` provides memory locked segments through the standard system facilities that are recognizable by the OpenCL implementation as memory locked. Furthermore, `Hybrid` requires the use of an OpenCL context only for the creation of the memory buffer while the actual allocation and memory locking are performed by standard library functions and they can overlap with OpenCL context creation.

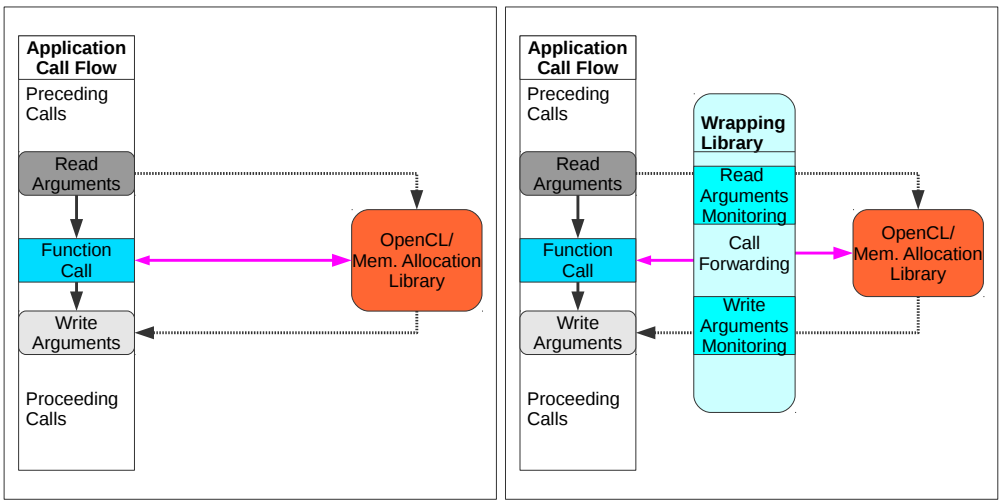
4.4.2 Platform Characterization Procedure

Two micro-benchmarks are used to determine memory allocation times and host-device communication rates. Both benchmarks are used to investigate a large range of allocation and data transfer sizes, respectively.

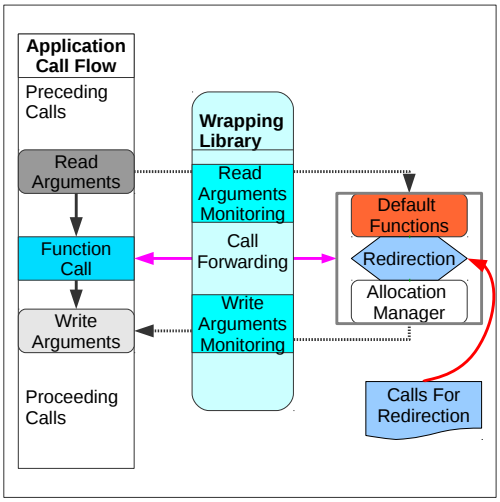
The `allocation micro-benchmark` investigates the overhead times for the allocation policies described above. Allocation sizes of interest range from 1KB to 767MB.

The `communication micro-benchmark` investigates the host-device communication rates where the used host memory is allocated with every available policy. The benchmark investigates the communication rates in both directions, from the host to device and from the device to host. The benchmark produces statistics for a number of concurrent data transfers ranging from one to four with transfer sizes ranging from 1KB to 150MB per transfer.

We perform curve fitting [22] on the collected data delivering performance estimation functions for both allocation times and communication rates. Both functions have a single input, the allocation and data transfer sizes respectively and they provide performance estimation in constant time.



(a) Standard Call. (b) Call with Tracing.



(c) Call with Optimization.

Figure 4.4: Function Call Intercepting. This figure shows how our framework intercepts the standard call procedure (Figure a) for the support of application tracing (Figure b) and runtime optimization (Figure c).

4.5 Application Tracing

To best exploit the allocation/communication properties of a platform, we need knowledge of application behavior. Our tool monitors the application execution by capturing and recording every call to the OpenCL runtime and memory allocation functions. Ideally, application tracing should be embedded in or attached to the OpenCL library. Unfortunately, the OpenCL library is distributed as closed source with no attachment mechanism. To overcome this, we developed a lightweight wrapping library that in-

tercepts calls to the OpenCL functions and the memory allocation functions defined in C/C++ and POSIX standards. Figure 4.4 (a) and (b) show how tracing is achieved.

Normally, as Figure 4.4a show, the application makes a call, passes the read arguments onto the stack and performs the call. On return, the write arguments have been updated. Figure 4.4b shows the call procedure when function calls are intercepted by our wrapping library. The read arguments are passed on the stack and the call takes place as before. Now, the execution control moves to our tracing code instead of the original function definition. The tracing library collects the read arguments and forwards the execution to the function definition. On function completion, the execution returns to our tracing code, that collects the write arguments and returns the execution control to the application.

4.5.1 Call Trace

The runtime application behavior is captured as a compressed `call trace`. The trace stores every application call to OpenCL and memory allocation functions along with data dependencies of the calls (read and write arguments of call). The trace also stores the overall application execution time and the execution times of host-device communication and kernel computation operations.

Two efficient data structures, called `Call` and `Data` represent the traced information for function calls and data objects, respectively.

Function Call Representation: An instance of `Call` stores information for a single function call. It contains the name of the function, the thread ID that performed it and its execution time if the function is a kernel computation or communication operation.

Data Representation: A `Data` instance stores information for a data object that is a read or write dependency for one or more function calls. Supported data types are the scalar types defined in C/C++ standards [46], abstract types of OpenCL, raw memory segments and arrays of the previous types. If a Data instance represents a scalar data object, it contains a copy of the actual data. If it represents a non-scalar, such as a raw memory segment or an OpenCL Memory object, it contains a pointer to the data object location and a unique ID provided by an SSA (Static Single Assignment) mechanism.

Non Scalar Data Versioning: We use SSA[24] representation to track non-scalar data objects during the application execution. After a non-scalar object definition, where the object is firstly created, or after its update by a function call, a new Data

```
1 segm=malloc(SIZE);
2 clSetKernelArg(kernel, 0, sizeof(void *), (void *)&buf);
3 for(i=0; i<n; i++)
4 {
5   do_smth(&segm);
6   clEnqueueWriteBuffer(cqueue, buf, 1, 0, SIZE, segm,0,NULL,NULL);
7   clEnqueueNDRangeKernel(cqueue, kernel, DIM, GWO, GWS, LWS, 0, NULL, NULL);
8   clEnqueueReadBuffer(cqueue, buf, 1, 0, SIZE, segm, 0, NULL, NULL);
9 }
```

(a) Sample OpenCL Code.

Call	Def	Use
malloc	s0	
karg	k1	b0,k0
wbuffer	b1	b0,s0,q0
kexec	b2	b1,k1,q0
rbuffer	s1	b2,s0,q0
wbuffer	b3	b2,s1,q0
kexec	b4	b3,k1,q0
rbuffer	s2	b4,s0,q0
...	n-3 loop iterations	...
wbuffer	b2n+1	b2n,sn,q0
kexec	b2n+2	b2n+1,k1,q0
rbuffer	sn+1	b2n+2,s0,q0

(b) Performed Function Calls.

Call	Def	Use
malloc	s0	
karg	k1	b0,k0
wbuffer(#n)	b1	b0,s0,q0
kexec(#n)	b2	b1,k1,q0
rbuffer(#n)	s1	b2,s0,q0

(c) Compressed Call Trace.

Figure 4.5: Call Trace Generation and Compression. Figure (a) shows a simple code segment where a set of OpenCL functions is executed *n* times. Figure (b) shows how the OpenCL calls are captured by the wrapping library over the loop iterations. Figure (c) shows the compressed trace after the completion of the code execution.

instance is created and stores a pointer to the data address and a unique ID provided by the SSA mechanism. Data instances that refer on different updates of the same non-scalar object have a pointer to the same data address but they have different SSA IDs. In addition, by tracking instead of copying non-scalar data objects we keep the tracing overhead to negligible levels.

4.5.2 Trace Compression

The input or problem size of an application execution may affect the number of executions for an OpenCL operation or group of operations. This could lead to huge call traces that vary among the different problem or input sizes. We solve this by detecting repeated calls to single functions or group of functions and merging the calls.

For trace compression, each call is represented as a production of the regular expression. We consider each call as a string, where the name of called function is followed by the SSA name of its non-scalar data arguments its return type if not a scalar.

Our compression technique is performed on the fly at run-time. Whenever a new call is added on trace, our compression algorithm checks for repeated calls to the same function or group of functions. Two calls are merged if they refer to the same function and the SSA IDs of their non-scalar arguments refer to the same objects. Merging between two groups of calls requires an one to one merging of the calls of the two groups. When we compress, we use the lowest SSA IDs. Our trace compression approach relies on the merging and transition labeling algorithm provided by [10]. We use a DFA based algorithm for the matching of the calls we merge. The effect of our compression technique is presented in Figure 4.5.

In the code of Figure 4.5a, after allocating a memory segment, the code copies the segment contents to a device buffer, computes a kernel and copies the buffer data back to the segment. This group of operations is performed n times due to the loop. Figure 4.5b provides the raw sequence of the function calls as they are performed during code execution. Each call entry contains the function name and its non-scalar data dependencies grouped as Definitions(Writes) and Uses(Reads). Figure 4.5c shows the resulting compressed trace. Our compression technique merged the repeated calls to the group of *clEnqueueWriteBuffer, clNDRangeKernel* and *clEnqueReadBuffer* functions to three final call entries. The full trace contains five calls, including the two calls that are not included in the loop.

$$\frac{\text{Cumulative Host-Device Communication Time}}{\text{Cumulative Device Computation Time}} \geq 0.1$$

Figure 4.6: Optimization Eligibility Heuristic. The application analysis considers an application as optimization eligible only if the communication overhead is comparable or higher to the computational workload of the device. The *dispatch ratio* (Cumulative Host-Device Communication Time / Cumulative Device Computation Time) is required to be greater or equal to 0.1.

Trace compression is critical for the host-device communication optimization. In practice, after compression, the compressed trace is less than 2 Kbytes. Our optimization technique requires that the trace provided by the application characterization and the one generated during the run-time optimization to be the same. If the two traces are different, the optimization is not applicable and the default scheme is used. Our experiments show that trace compression produces matching traces across all programs within two benchmark suites.

4.6 Application Analysis

Once we have the compressed trace, we analyze it for optimization opportunities. Memory allocation calls may be redirected at runtime from the standard allocation policy to the one that leads to the highest transfer rates. However, an application may perform dynamic memory allocations that are never involved in host-device communication and are only used in host operations. This type of allocation cannot benefit from our scheme and may indeed incur overhead if redirected. Here, we describe the application analysis that firstly checks if the optimization is worthwhile and then detects the memory allocations that should be redirected.

An optimization for the host-device communication is meaningful in cases that the communication overhead is comparable or higher than the computational workload dispatched on the device, otherwise the communication improvement will not lead to significant application speedups. Our application analysis checks if the optimization is worthwhile for an application with the heuristic of Figure 4.6. If the *dispatch ratio*, defined as the ratio of the Cumulative Host-Device Communication Time over the Cumulative Device Computation Time, is greater or equal to 0.1, the application is

considered optimization eligible. The value of 0.1 is specified after experimentation and guarantees significant performance gain regardless the application.

If the application is optimization eligible, the algorithm 4.2 operates on the compressed call trace of the application and detects the dynamic memory allocations that are involved in OpenCL communication operations. The algorithm traverses the call trace and for each call that performs a host-device communication operation, it retrieves the used memory segment. The algorithm then retrieves the first `Data` instance that refers to the segment through SSA. After it annotates the memory allocation call that created the data object as optimization candidate.

The output of the algorithm is the set of memory allocations that should be redirected by the runtime optimization.

```
1 for each c in the Call Trace
2  if c is a host-device communication that involves a memory segment s
3   retrieve s', the first state of s (through SSA)
4   retrieve co, the creator (allocation call) of s'
5   annotate co as optimization candidate
```

Algorithm 4.2: Allocation Detection Algorithm. The algorithm operates on the Call Trace of the application and detects the dynamic memory allocations that are used in host-device communication operations.

4.7 Runtime Optimization

Our enhanced execution environment controls the host-device communication optimization. It uses the available memory allocation policies and performance estimation functions provided by platform characterization. The optimization redirects the memory allocations indicated by the application analysis from the default allocation policy to an allocation manager which decides the best policy to be used.

Request Interception: The redirection is achieved by intercepting the standard call procedure with a technique similar to the one of application tracing (Section 4.5). Figure 4.4c illustrates the call redirection. While the application is executed in the enhanced environment, a call trace is constructed with a procedure identical to the one of application tracing. Whenever a new allocation call is added to the trace, the optimization checks if it is indicated by the application analysis for redirection. If not, the call is served by the default memory allocation functions. Otherwise the call is redirected to our allocation manager.

Trace Comparison: As described in Section 4.5, the optimization technique requires the call trace generated by the application tracing phase and the one generated during the application optimization to be the same. Trace compression preserves this property for different sized inputs but not for different application behaviors. In case that the application presents a behavior different than the expected one, where it constructs a different call trace, the enhanced environment will detect it. The enhanced environment checks the function names and non-scalar data arguments of the calls it encounters. In the case that they differ from the application analysis data, the optimization is deactivated. The remaining of the execution performs conservatively and the application completes its execution safely.

4.7.1 Memory Allocation Manager

It has direct access to the available allocation policies and serves the redirected allocation requests. It serves each allocation request by selecting the policy that gives the highest performance for host-device communication. The decision relies on the performance estimation functions and the size of the allocation.

Depending on the allocation times of an allocation policy, the manager either uses the policy directly or leverages a user-space memory allocator for it. An allocator uses directly a policy to allocate a large memory segment during environment setup and performs its own allocation algorithm in user-space. The allocation algorithm relies on a Red-Black tree for efficient indexing of free memory chunks.

Policy Selection: During the environment setup, the allocation manager checks the allocation times of the policy that leads to the highest communication performance across varying transfer sizes. If it has high allocation times, a user-space memory allocator is initialized. If two or more policies lead to similar communication performance, the environment chooses the one that does not require allocator and deactivates the others. If all of them require one, it chooses one of them.

Allocator Initialization: The memory allocator initialization is possible to overlap with the OpenCL context creation, if the memory allocation operation of the policy can operate partly or fully without an OpenCL context. `Standard with Locking` can overlap completely with the context creation. In case of `Hybrid`, the time consuming part of the allocation overlaps, as the memory allocation and locking is performed with standard functions. The enhanced environment tends to enable allocators with policies that can overlap with OpenCL context creation.

Platform	CPU	RAM	GPU
GTXplatform	Intel i7 X990 3.47GHz	12GB 1333MHz	NVIDIA GTX 580
AMDplatform	Intel i7 X990 3.47GHz	8GB 1333MHz	ATI Radeon HD 5970
K20platform	Intel i7 3820 3.60GHz	8GB 1333MHz	NVIDIA Tesla K20c

Figure 4.7: Overview of the Evaluation Platforms.

Platform Limitations: Resource limits may be specified by the Operating System or the OpenCL runtime when the environment uses special memory allocation policies. Our work respects those limitations. In the case that a memory allocation policy fails to serve more requests, the environment falls back to standard memory allocation policy.

4.8 Experimental Setup

Our host-device communication optimization approach is evaluated against two benchmark suites on three different platforms

4.8.1 Platforms

We use 3 platforms: the `GTXplatform`, `AMDplatform` and `K20platform` as shown in figure 4.7. Each platform has an Intel i7 multicore and run Linux. The `GTXplatform` has an NVIDIA GPU of `Fermi`[110] architecture. The `AMDplatform` has an AMD GPU of `Evergreen`[3] architecture. The last platform, `K20platform`, has an NVIDIA GPU of `Kepler`[80] architecture. The GPUs are connected with the main system through PCI Express interconnects of version 2.0, 2.1 and 2.0 respectively.

4.8.2 Benchmarks

We use both the Parboil [101] and Rodinia [18] benchmark suites for evaluation. From Parboil, we consider two versions of each benchmark, a `generic` one and a specialized version, `nvidia` tuned for NVIDIA GPUs.

For each benchmark we consider two available inputs, one small and one large, to evaluate our approach for different problem sizes. In certain cases, `sad`, `streamcluster` and `mri-gridding`, only one data set is available, The `backprop` benchmark of Rodinia and the NVIDIA version of `bfs` of Parboil both crash with segmentation faults

and are excluded.

We also exclude the `bfs[r]` and `streamcluster` from our evaluation. These two Rodinia benchmarks both have hard-coded special memory allocation policies which prevent portability and evaluation across platforms.

4.9 Results

In this section we evaluate our host-device communication optimization on the Parboil and Rodinia benchmark suites on the 3 platforms. For those benchmarks which are eligible for optimization, the reported speedups are significant; ranging from 1.25x for small data zises on the `AMDplatform` to 1.51x on the `GTXplatform` for large datasizes.

Section 4.9.1 discuss in detail the evaluation results on the `GTXplatform`. It provides data for the dispatch ratios of the benchmarks, analyzes the speedup results and provide execution time breakdowns that give an insight of the benchmark behavior in both the standard and enhanced environments. This is followed by section 4.9.2 which presents the results of our optimization on `AMDplatform` and `K20platform`.

In section 4.9.3 we evaluate our optimization on the NVIDIA tuned version of the Parboil benchmarks and show how our technique has additional impact on performance. This is followed in section 4.9.4 by an analysis of the allocation policies selected by our scheme. In section 4.9.5, we compare our optimization technique against potential naive approaches.

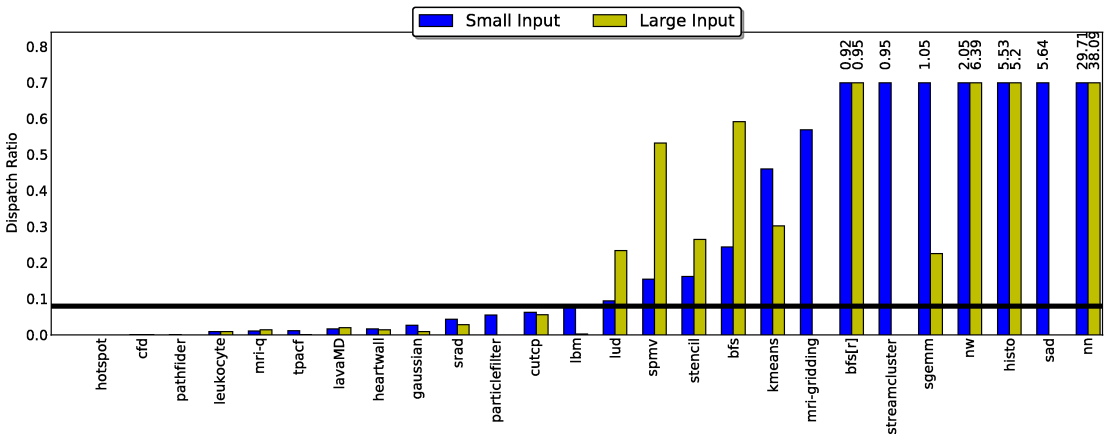
4.9.1 Results on NVIDIA GTX 580

Dispatch ratio:

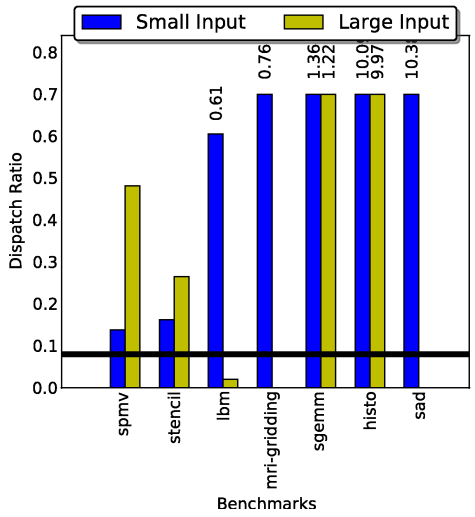
The optimization eligibility of a benchmark is dependent on the `dispatch ratio` (the ratio of Cumulative Host-Device Communication Time over the Cumulative Device Computation Time) of its small input. If the value is greater or equal to 0.1 the optimization is applied to the benchmark (Section 4.6). The use of this heuristic guarantees that the optimization will improve the benchmark execution time significantly.

To illustrate this, Figure 4.8a presents the dispatch ratios for the two inputs of each benchmark. The benchmarks are ordered by their dispatch ratio for their small input, starting from the benchmark with the lowest value.

Providing both dispatch ratios per benchmark is *not* required by the eligibility heuristic of application analysis but is useful in interpreting speedup results. For bench-



(a) Generic Benchmark Version (generic)



(b) NVIDIA Tuned Version (nvidia)

Figure 4.8: Benchmark Dispatch Ratio. The figures show the dispatch ratios for the benchmarks of both Parboil and Rodinia suites. Each benchmark has two bars representing the dispatch ratio for two different inputs. Benchmarks whose dispatch ratio is equal or greater to 0.1 for their small input are considered optimization eligible. Figure (a) provides dispatch ratio values for the generic version of benchmarks. Figure (b) shows the dispatch ratio for NVIDIA version of Parboil benchmarks that are originally optimization eligible.

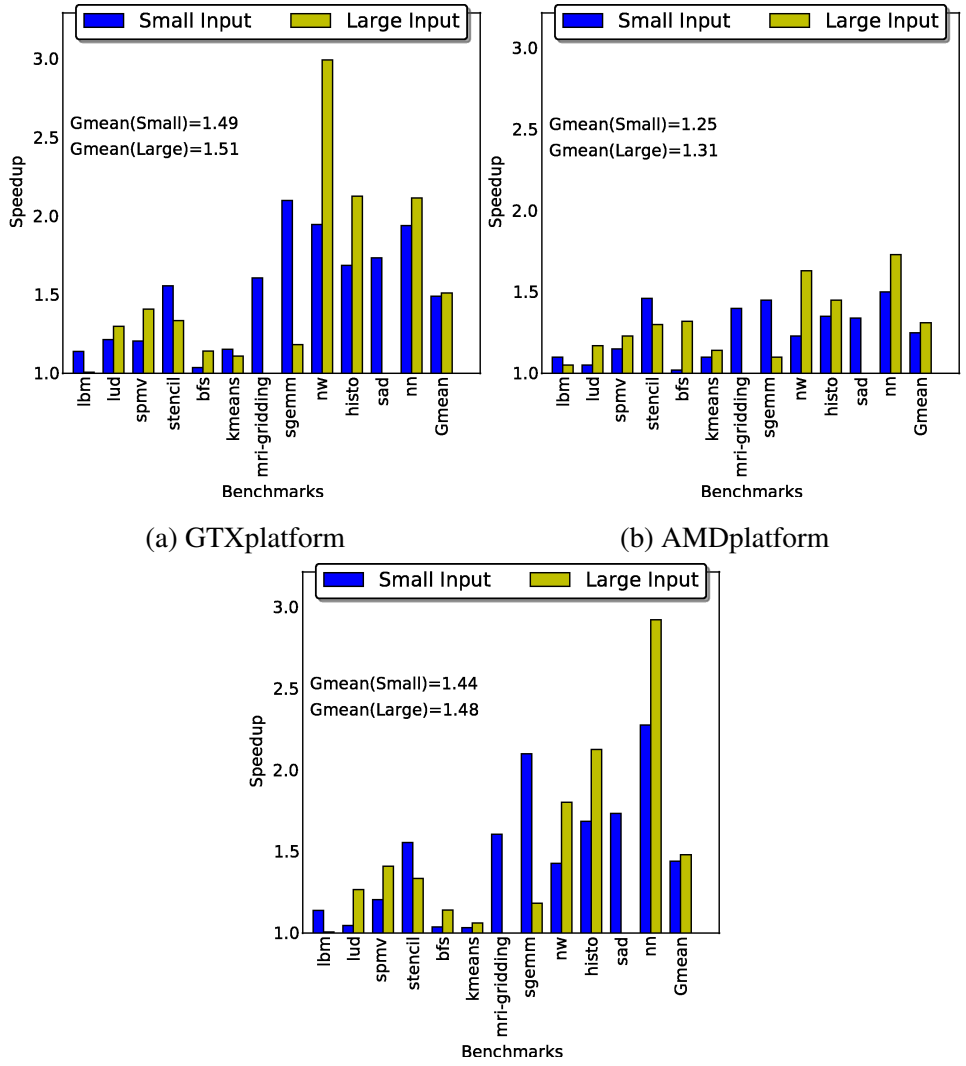
marks such as bfs and nn the dispatch ratio increases on the large input. For bfs, dispatch ratio increases from 0.25 to 0.55 and for nn from 29.71 to 38.09. For others such as kmeans and sgemm the dispatch ratio decreases on the large input. The ratio of kmeans decreases from 0.45 to 0.32 and for sgemm from 1.05 to 0.2. Benchmarks such as histo has relatively similar dispatch ratios for both inputs. The ratios of histo are 5.53 and 5.2 for its small and large inputs as computation scales with communication for this benchmark.

Speedups: Figure 4.9a shows the speedups achieved by the execution of benchmarks in our enhanced environment compared to their execution in the standard environment. The speedups range from 1.05x to 3x. Benchmarks that have dispatch ratios proportional to the input size, such as nw and nn, present higher speedups for the large input. For nw and nn the speedup increases by 1.0 and 0.2, respectively. Benchmarks that have dispatch ratios inversely proportional to the input size, such lbm and sgemm have lower speedups for the large input.

For histo, our optimization gives significantly higher performance with the large input despite a dispatch ratios similar to the smaller size, This is because the large input can be improved by a special policy. For large input, the memory manager allocates segments with the Hybrid policy, which leads to the lowest communication overhead. The execution with the small input requires only small size transfers where a special policy is not beneficial and the manager uses the Standard policy(Section 4.9.4).

Execution Breakdown: Figure 4.10a presents the execution time breakdowns for the benchmarks running in both the standard and our enhanced execution environment for the large input sizes. Each benchmark has a pair of bars. Each bar shows a stack which gives the total time spent executing the program broken down into subcomponents: kernel execution time, communication time between host and device and CPU + Sync time. The optimization reduces the host-device communication times and in many cases the CPU + Sync times. This time includes the delays caused by synchronizations. Synchronizations are performed either explicitly by synchronization calls or implicitly by the ordering and synchronization of OpenCL command queues and the OpenCL implementation. The reduction of host-device communication times leads to the reduction of those delays too.

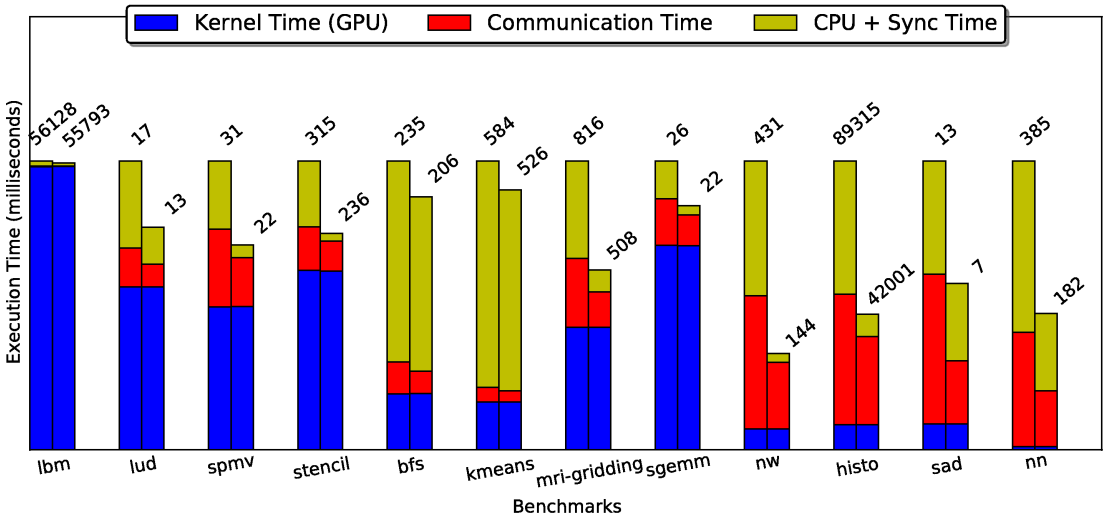
Benchmarks that have high dispatch ratios and speedups tend to spend significant parts of their execution for communication and the optimization reduces massively the communication times. Benchmarks such as nw, histo and sad present critical reduction of their communication and cpu+sync times. Benchmarks such as lbm and



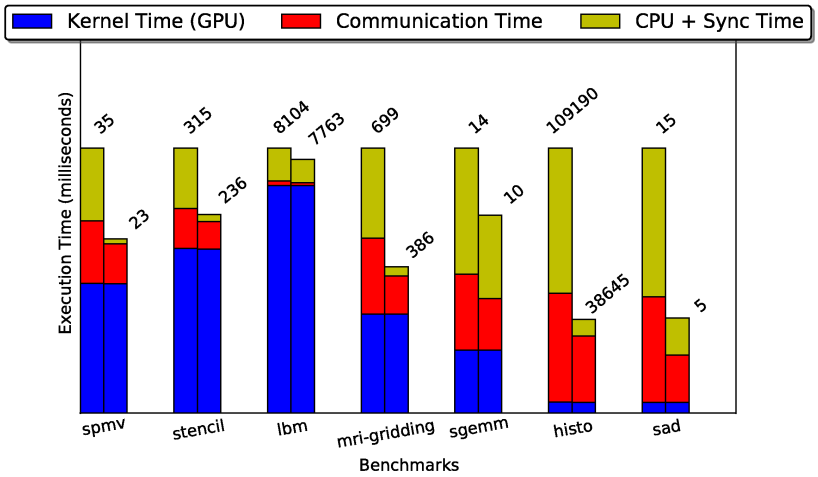
(a) GTXplatform (b) AMDplatform

(c) K20platform

Figure 4.9: Benchmark Speedup. The figure shows the speedup for the optimization eligible benchmarks on the three platforms. It compares the overall benchmark execution times of benchmarks running on the standard execution environment of OpenCL with the times of our enhanced execution environment that enables the host-device communication optimization. It provides results for two input sets and the resulted geometric means.



(a) Generic Benchmark Version (generic)



(b) NVIDIA Tuned Version (nvidia)

Figure 4.10: Execution Time Breakdown. This figure shows the execution time breakdowns for the benchmarks running on the standard execution environment (left bar) and our enhanced execution environment (right bar) which enables our optimization. The data refers to the large input of benchmarks.

lud which have less communication, show a lower scale of performance improvement. The breakdown for the smaller input has a similar structure.

4.9.2 Results on AMD Radeon HD 5970 and NVIDIA Tesla k20c

For the AMDplatform, the speedups of figure 4.9b are significantly lower than the speedups of the platforms with NVIDIA GPUs. This is because of the AMD OpenCL implementation. It preallocates memory locked segments and uses them as intermediate buffers in host-device communication[3]. In addition, the size of the available memory locked segments is limited. For many benchmarks such as histo and nn, this amount of memory is insufficient and the allocation manager is forced to switch back to the use of the default memory allocator, when OpenCL policy runs out of memory. The application completes its execution properly but the optimization opportunity is lost. However, the speedup results on the platform have similar trends to the NVIDIA platforms. Benchmarks such as nw and nn have high speedups and benchmarks such as bfs and kmeans present lower speedup levels.

Figure 4.9c shows the speedups achieved by our optimization on K20platform. There is similar speedup levels as the GTXplatform with the exception of the nw and nn benchmarks. The nw kernel code runs less efficiently on K20platform and it cuts down the relative impact of communication overhead reduction. In contrast, the nn kernel code is faster on K20platform.

4.9.3 Tuned Version of Parboil for NVIDIA

As programmers tune their applications, how does this affect our approach? To answer this, we evaluate our host-device optimization with the tuned version of Parboil benchmarks for NVIDIA GPUs.

Figure 4.8b shows the dispatch ratios of the benchmarks, which are now increased. The ratios for sad and histo are doubled. The sgemm and lbm benchmarks present highly increased ratios for their large and small inputs, respectively. The remaining of the benchmarks present lower increases with the exception of stencil that remains on the same levels.

Figure 4.11 shows the optimization speedups for the NVIDIA version of benchmarks. The optimization now leads to higher speedups for all the benchmarks with the exception of stencil, which has the same speedup as its generic version. The sad and histo benchmarks obtain the largest speedup increase. Sad increases from a speedup

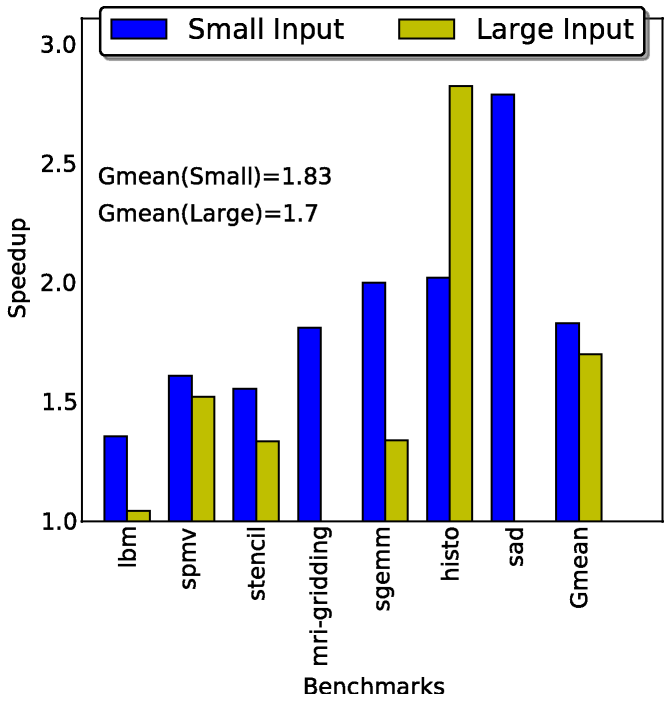


Figure 4.11: Benchmark Speedup (NVIDIA tuned version).

1.7x of its generic version to 2.8x. `Histo` delivers speedups of 2x and 2.8x for its inputs, which are increased by approximately 40% and 60% in comparison to its generic version. The `lbm`, `mri-griding` present a lower speedup increase of approximately 20%. In the case of `sgemm`, we observe approximately the same speedup for the small input and a speedup increase of approximately 20% for the large one. In case of `spmv`, the small input presents a speedup increase from 1.2x to 1.6x and its large one presents a speedup increase of 10%.

Surprisingly, in the case of `histo`, its execution time is much slower than the generic version, despite it being tuned for the GPU. This happens in both the standard execution environment and our enhanced execution environment and it is caused by the benchmark implementation. Figure 4.10b shows the execution time breakdowns for the NVIDIA version of the benchmarks. In this case, the optimization leads again to the reduction of communication and cpu+sync times. The impact on the `K20platform` is similar.

4.9.4 What policy to use

At the heart of our technique is the benefit gained form different allocation policies. In this section we analyze the allocation time overhead and host-device communica-

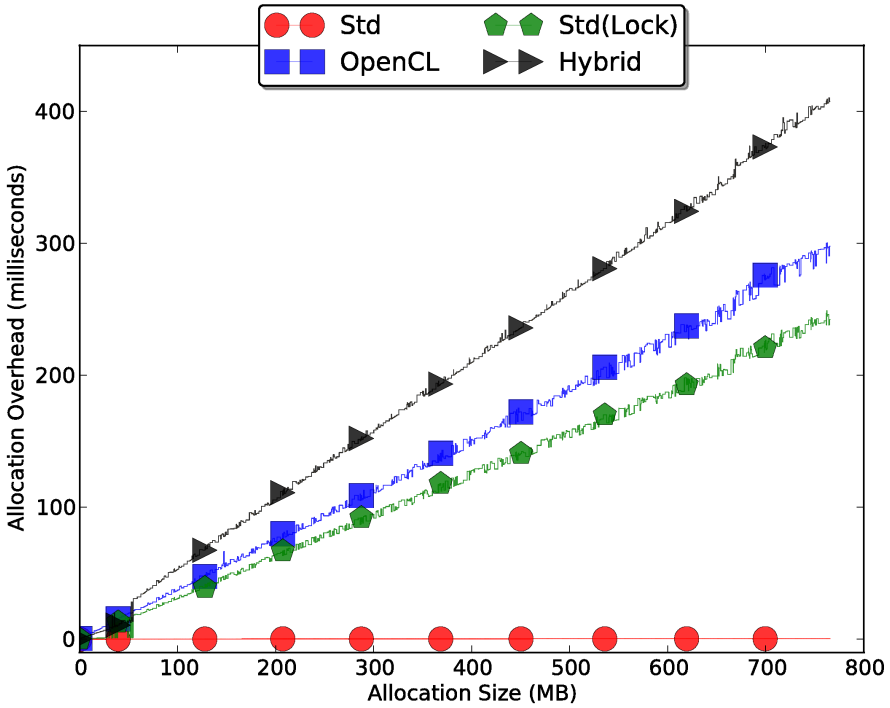


Figure 4.12: Allocation Overhead (in milliseconds) for the four policies for a range of allocation sizes on `GTXplatform`. Std, Std(Lock), OpenCL and Hybrid refer to the Standard, Standard with Locking, OpenCL and Hybrid policies, respectively. Lower is better. The circle and polygon shapes are meant for easing readability on black and white printouts.

tion rates for each policy. We provide detailed statistics for the `GTXplatform` and we describe the capabilities of the three platforms.

Figure 4.12 shows the memory allocation cost of the four policies for a range of allocation sizes on the `GTXplatform`. The `Standard` (`Std`) policy shows negligible allocation times in comparison to the other policies. `Standard with Locking` (`Std(Lock)`) has notably higher cost but remains significantly lower than the `OpenCL`. The last policy, `Hybrid`, has the highest overhead.

Figure 4.13 shows the communication overhead of the four policies for communication in both directions on the `GTXplatform`. Here, lower values denote higher communication performance. `HTD` denotes a host to device transfer and `DTH` the opposite. It is clear that both `OpenCL`(`OpenCL`) and `Hybrid`(`Hybrid`) lead to the lowest communication cost for the majority of transfer sizes in both communication directions. In contrast the `Standard`(`Std`) and `Standard with Locking`(`Std(Lock)`) lead to notably higher communication overheads. Thus the fastest allocation time comes at the expense of greater communication cost.

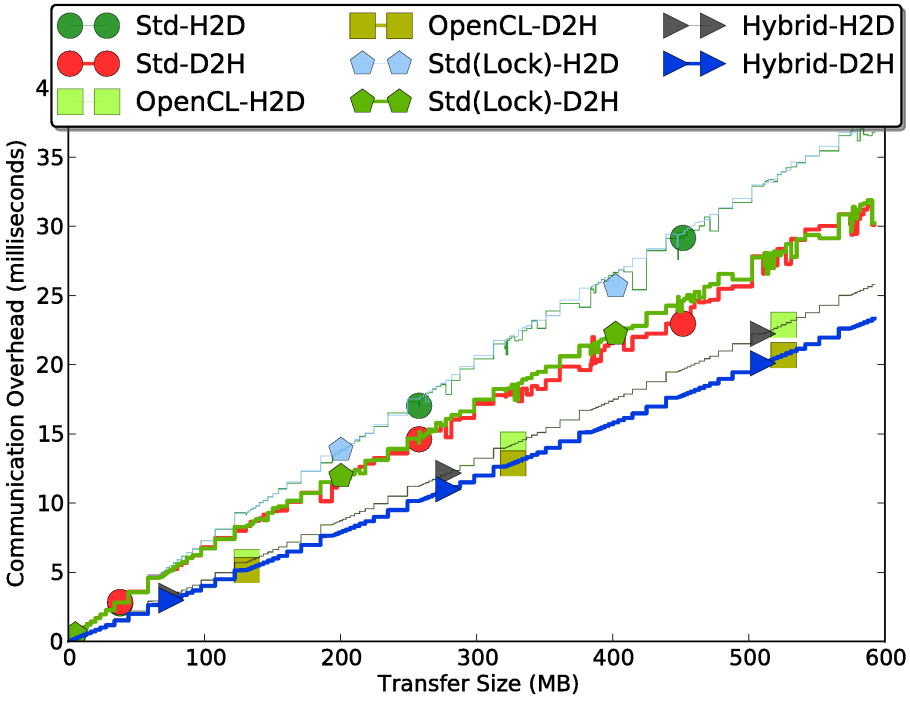


Figure 4.13: Communication Overhead (in milliseconds) for the four policies on `GTXplatform`. Std, Std(Lock), OpenCL and Hybrid refer to the Standard, Standard with Locking, OpenCL and Hybrid policies, respectively. H2D denotes a data transfer from the host memory to device memory and D2H the opposite. Lower is better. The circle and polygon shapes are meant for easing readability on black and white printouts.

The `OpenCL`, `Hybrid` and `Standard with Locking` policies provide memory locked segments, which have their memory pages pinned permanently in main memory and the GPU driver should take advantage of it. However, only `OpenCL` and `Hybrid` lead to peak communication rates and lower overheads. The reason is that the NVIDIA OpenCL implementation cannot recognize a segment provided by `Standard with Locking` as memory locked. In addition, it should be reported that for low transfer sizes, all the policies lead to similar communication overheads.

GTXplatform: On this platform the enhanced execution environment uses `Standard` for allocation of small segments and `Hybrid` for larger sizes. It chooses `Hybrid` instead of `OpenCL` because the initialization of the required user-space allocator with `Hybrid` overlaps with the OpenCL context creation.

AMDplatform: Here the policies that provide memory locked segments lead to a lower scale of performance improvement. This potentially happens because of OpenCL implementation internals. The AMD implementation allocates memory locked seg-

ments and uses them as internal buffers during host-device communication. It may also lock memory segments of the application under specific circumstances[3]. This implementation makes the policies with memory locking less effective. The enhanced environment uses the `Standard` policy for low sizes and the `OpenCL` for higher.

K20platform: Here the behavior is similar to `GTXplatform` with similar communication overheads. NVIDIA Tesla K20c supports PCI Express version 3.0, however the motherboard of `K20platform` supports only the second generation of PCI Express and the GPU adapts to it. The enhanced environment uses the same policies as for `GTXplatform`.

Summary: This evaluation has shown that selecting the correct allocation policy based on platform and communication size can have significant performance impact across all 3 platforms. It gives on average a speedup of 1.51, 1.31 and 1.48, respectively. In certain cases, our approach leads up to a factor of three times improvement.

4.9.5 Comparing against a naive approach

Our optimization leverages user-space memory allocators for policies with high allocation times. In that way, it reduces prohibitive allocation overheads. Here we compare this approach against an alternative naive scheme where all allocation policies are used directly without the use of user-space allocators. Using this approach increases execution time by 5% to 40%. It could also be argued that all memory allocations requests should be treated equally avoiding the need for application characterization. In fact less than 5% of the total memory allocations across all the benchmarks are optimization candidates. Using a high overhead allocator for the remaining 95% is prohibitively expensive. Our approach is clearly important for performance.

4.10 Summary

In this chapter, we presented a portable and transparent optimization for the reduction of host-device communication overhead of OpenCL applications. Our technique is platform and application unaware and does not require modification or recompilation of the application source code. The optimization was applied to 12 existing OpenCL benchmarks where it gives on average speedups of 1.51, 1.31 and 1.48 for three evaluation platforms. In certain cases, it leads up to a factor of three times improvement.

The next chapter presents the design and implementation of an acceleration layer

for heterogeneous resources. It enables central management of accelerator resources and fine-grained, mixed-vendor accelerator sharing. It integrates with the existing multi-tasking and user-space virtualization facilities of the commodity Linux OS.

Chapter 5

Heterogeneous Acceleration Layer

This chapter presents a secure, user-space virtualization layer that integrates the accelerator resources of a system with the standard multi-tasking and user-space virtualization facilities of commodity Linux OS. It targets heterogeneous commodity systems found in data center nodes and requires no modification to the OS, OpenCL or application. It eliminates high setup overhead, enables fine-grained sharing of mixed-vendor accelerator resources and provides resource and platform aware scheduling. The average throughput improvement across workloads and mixed-vendor platform configurations varies from 1.29x to 3.87x speedup over existing schemes. Our approach outperforms both vendor accelerator sharing facilities and message passing solutions.

The remainder of the chapter is organized as follows. An introduction of the challenges and contributions of this chapter are given in section 5.1. Section 5.2 describes our motivation for PALMOS. Section 5.3 provides a high-level overview of PALMOS. Sections 5.4, 5.5 and 5.6 present the key components and features of PALMOS. Section 5.7 describes our security control. We present our experimental setup and evaluation in sections 5.8 and 5.9, respectively. Section 5.10 discusses chapter summary.

5.1 Introduction

Accelerators, such as Graphic Processing Units (GPUs), have proved to be popular components of modern heterogeneous systems. They provide the potential for low-cost, high-performance computing for highly parallel workloads. This move towards heterogeneity has been mirrored by the development of parallel programming languages such as

CUDA[81] and OpenCL[58] which is portable across accelerators and standard multi-cores.

However, there is no further integration across the system software stack. Each accelerator is still viewed as a co-processor managed by the application; there is no transparent Operating System (OS) support for sharing accelerator resources between applications and users. There is no scheduling control for multi-accelerator systems. Currently, an accelerator is either exclusively dedicated to an application or supports limited sharing via vendor drivers where the sharing preconditions are non-obvious and lack fine-grained control. Such an approach may be sufficient for traditional HPC and single application systems, but introduces significant issues for multi-tasking systems.

There are a number of issues that are barriers to general use of heterogeneous systems. Firstly, accelerator sharing is either limited or not supported depending on the vendor. There is no support for mixed-vendor accelerator sharing between applications. Secondly, there is no OS-level resource isolation support for heterogeneous software using accelerators. Applications ignore the presence and activity of other applications and their accelerator usage which leads to resource contention. Thirdly, application setup times for heterogeneous software are currently prohibitive. This introduces a significant overhead for multi-tasking and virtualized environments where multiple applications co-exist. Finally, as applications run on shared memory platforms with multiple physical nodes, data placement and the selection of appropriate CPUs and accelerators can significantly affect the performance of an application or the system.

In this chapter, we introduce `PALMOS`, a user-space virtualization layer that integrates the accelerator resources of a system with standard OS multi-tasking and enables OS-level virtualization for heterogeneous software. Our design reduces setup overhead, enables inter-vendor accelerator sharing and provides resource aware scheduling for heterogeneous applications. Our approach exclusively relies on the interfaces of OpenCL and standard system libraries. It remains compatible with existing software and systems. It does not require **any** modification of the OS, OpenCL library or application.

PALMOS targets standard heterogeneous systems found in data center nodes. Datacenter nodes support large numbers of diverse user tasks which need both performance and secure execution. While there is support for communication and task scheduling between nodes such as Quasar[26], or Whare-Map[72], there is no appropriate support on node level, specially for systems with accelerators.

Hypervisors such as Xen[6]or Kvm[61] efficiently support OS virtualization but there is an upcoming need for more lightweight approaches [5][99][2][108] which provide virtualization and resource isolation for individual processes at user-space instead of full OS virtualization. Production technologies such as Docker [28] and OpenVZ [84] enable user-space virtualization but they focus on traditional applications and not on heterogeneous software. NVIDIA supports accelerator sharing between Virtual Machines [82]. However, it does not provide scheduling and resource sharing control or user-space virtualization. It is also vendor specific.

PALMOS performs accelerator management and sharing at user-space level, enabling virtualization for heterogeneous software. To summarize, this work makes the following contributions:

- A lightweight and secure virtualization layer for multi-user heterogeneous computing.
- Seamless system integration requiring no modification to existing applications, OS or accelerators.
- Fine-grain sharing and control of inter-vendor accelerator resources between applications and users.
- Resource and NUMA aware multi-tasking and scheduling on host and accelerators.

We evaluate PALMOS across different, mixed-vendor platform configurations and workloads. It improves average throughput from 2.1x to 3.87x speedup over existing schemes.

5.2 Motivation

This section provides a simple motivational example for this work. Consider figure 6.1 which shows the execution of four heterogeneous applications on a system with two accelerators.

On current systems, as shown in the left hand side of the figure 6.1, applications directly access the accelerators. Depending on the capabilities of the accelerators, some applications may fail to execute, stall or execute concurrently[80]. The OS has no

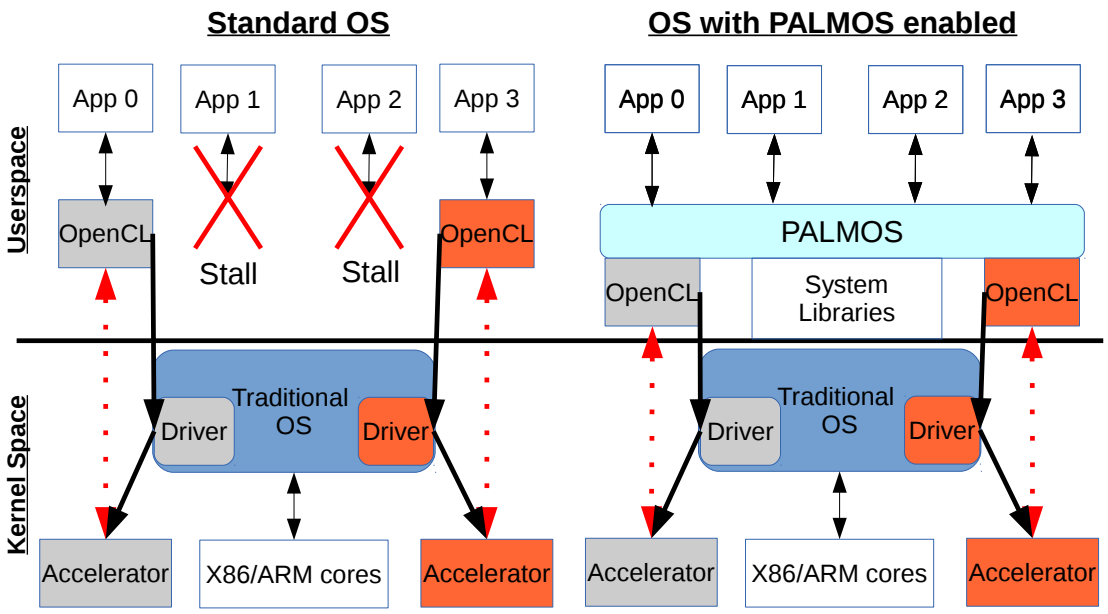


Figure 5.1: Motivation for PALMOS. Standard OpenCL accesses directly accelerator resources, suffers from large setup times and there is no resource sharing control. PALMOS enables fine-grained and inter-vendor accelerator sharing between multi-user applications, reduces setup overhead and performs resource aware scheduling.

knowledge about the accelerators and their capabilities and completely ignores memory transfers and scheduling on them. Everything is controlled by the application and there is no handling from a system perspective.

PALMOS, as shown in the right side of the figure, operates transparently as a user-space virtualization layer between the application and OS. Here, the applications do not directly access the accelerator resources. Instead, PALMOS controls scheduling on CPUs and accelerators and enforces fine grained inter-vendor sharing between the applications. Furthermore, it integrates smoothly with OS-level virtualization facilities. It operates transparently with standard application processes and applications running as part of containers built on the top of Linux namespaces[74].

5.3 Layer Overview

In this section, we provide a high-level overview of PALMOS, our user-space virtualization layer that integrates accelerator resources with commodity OS components. The design of the layer is portable, built on top of OpenCL and standard system libraries. The layer is transparent to both the OS and applications, requiring no modifications. It operates as a high priority user-space process which collects and manages

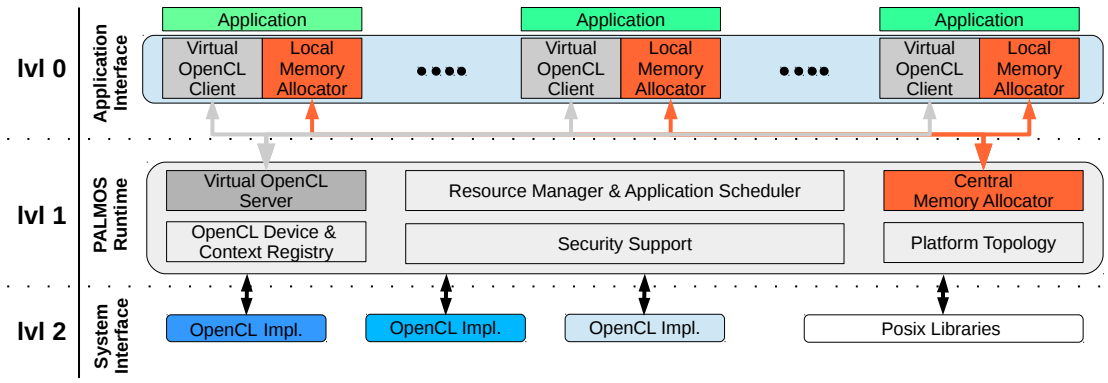


Figure 5.2: Overview of PALMOS. The layer design consists of three levels, the Application Interface (level 0), the PALMOS Runtime (level 1) and the System Interface (level 2), its connection to existing OS and runtimes. PALMOS operates transparently between existing applications, OS and OpenCL runtimes.

computation and communication requests from different applications and users. It then schedules them on accelerator resources.

PALMOS builds an environment where each application can safely assume that it exclusively uses the host and accelerator resources. The application performs ordinary operations, computes on the host, allocates memory and makes the ususal calls to OpenCL functions for accelerator use. However, these function calls are forwarded to PALMOS which decides, for each application, which host CPUs to use, how to allocate memory and which accelerator to exploit.

5.3.1 Key Design Choices

One of the primary goals of PALMOS is portability and transparency i.e. no change to existing applications, libraries or OS. This means that the APIs that connect an OpenCL application to hardware, OpenCL library calls and standard memory management, must be preserved.

The next design goal is to support efficient heterogeneous multi-tasking and accelerator sharing. PALMOS is a separate layer between the applications and OS that manages all the system resources and controls scheduling to improve system throughput. These design decisions, however, introduce two problems: *communication overhead* and *security*.

Both PALMOS and the applications are ordinary processes that reside in different address spaces. This preserves compatibility as no piece of software has to be changed or recompiled. In addition, each application executes in a secure, isolated environment.

A new challenge is therefore, how to achieve efficient inter-process communication and data sharing between PALMOS and its attached applications.

We solve this by using POSIX shared memory segments[83], segments of memory that are allocated independently of a single process and that are later mapped to the address spaces of the associated processes. This solution dramatically reduces overheads but raises security concerns that we address below and in more detail in section 5.7.

5.3.2 PALMOS Structured Design

Figure 6.5 gives an overview of PALMOS. The design consists of three levels. The Application Interface, level 0, enables the communication of application with PALMOS and guarantees transparency and portability. The APIs provided to applications are identical to those of OpenCL and system libraries. Level 1 is the heart of PALMOS; it provides inter-vendor accelerator sharing, scheduling control, environment setup, memory allocation and security. At level 2, the System Interface connects PALMOS with the existing OS and runtime libraries.

Level 0: The Application Interface: Applications are attached to PALMOS and interact with it without modifications. This is done via two interfaces (a) a Virtual OpenCL client that replaces the standard OpenCL library and (b) a Local Memory Allocator, provided by PALMOS, that replaces the standard memory allocator. Both interfaces are supplied as dynamic libraries that are preloaded.

Virtual OpenCL Client: It supports all OpenCL 1.2 functions and redirects any application call to them to the *Virtual OpenCL Server* of PALMOS instead of the actual OpenCL library. Virtual OpenCL is built on the top of shared memory for efficiency. We provide a detailed description of this mechanism in section 5.4.

Local Memory Allocator: This serves the memory allocation requests of the application. It is part of a two-level *"Inter-Space" Memory Allocator* which enables data sharing between the application and PALMOS, which are two distinct processes with distinct address spaces. The allocator leverages shared memory segments and an *Address Space Translation* mechanism to enable zero copy data sharing. Section 5.5 provides a detailed description.

5.3.2.1 Level 1: The PALMOS Runtime

This layer provides the core functionality of PALMOS including accelerator management, resource sharing control and improved scheduling.

Resource Manager & Application Scheduler: This monitors the execution of all applications and makes decisions about their scheduling on host and accelerators. It also takes care of data placement in multi-node shared memory, NUMA platforms. The scheduler receives operation requests via Virtual OpenCL and dispatches OpenCL operations to an accelerator based on its scheduling decisions. Section 5.6 provides further details.

Virtual OpenCL Server, Central Memory Manager: These are the PALMOS-side counterparts of the *Virtual OpenCL* and *Inter-Space Memory Allocator*. Sections 5.4 and 5.5 provide further details.

OpenCL Device & Context Registry: It stores accelerator specific information and OpenCL contexts. OpenCL contexts are now part of the PALMOS runtime relieving applications of high setup overheads.

Security Support: This addresses security within PALMOS. The PALMOS process and the application processes communicate through shared memory segments. In the absence of security control, there are potential threats. An application could send an invalid OpenCL call or attempt to communicate through a Virtual OpenCL Client it does not own. An application may also attempt to access data of a different application. This component prevents this by controlling the access to shared memory segments. Further details are provided in section 5.7.

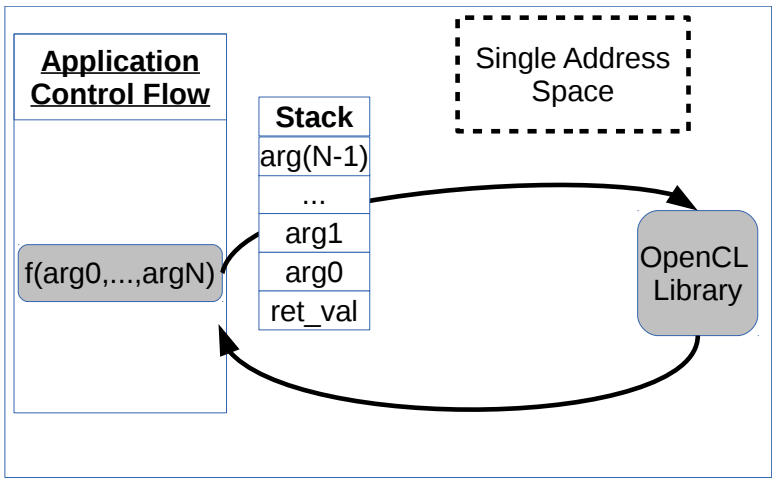
Platform Topology: This detects the topology of multi-node systems including the number of nodes, the size of local memories, the processor and accelerator locations.

5.3.2.2 Level 2: The System Interface

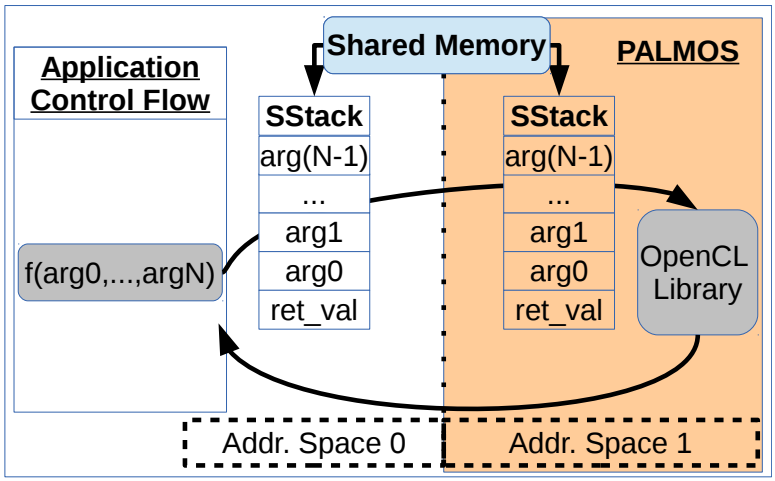
PALMOS exclusively depends on OpenCL, POSIX[83] and Linux system[67] libraries. OpenCL provides access and management of the accelerator resources while POSIX and Linux libraries are used for the management of standard CPU and memory resources, scheduling of application host code, data placement and security control.

5.4 Virtual OpenCL

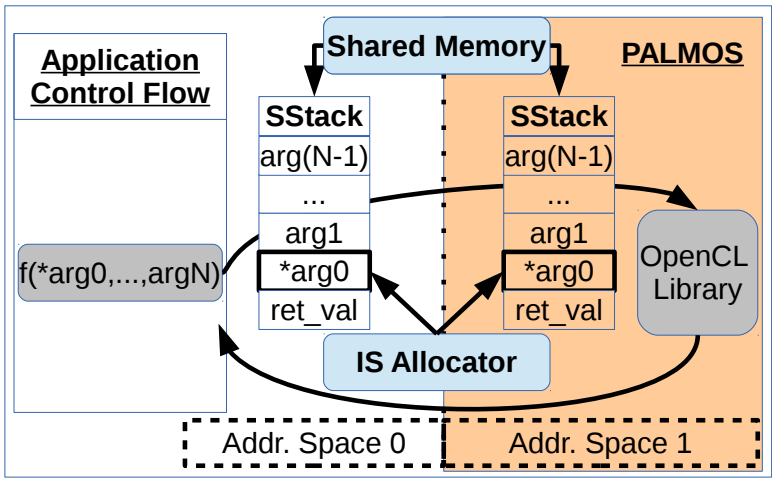
Virtual OpenCL forwards OpenCL operation requests to PALMOS. It is available to the application as a dynamic library that replaces standard OpenCL. It ensures transparency for the application, which operates as before without any modification or recompilation. Virtual OpenCL consists of (a) a client that is loaded by the application



(a) Standard OpenCL Call



(b) RPC OpenCL Call



(c) RPC OpenCL Call (IS M. Allocator)

Figure 5.3: Comparison of the function call convention of the standard OpenCL environment (a) with our Virtual OpenCL mechanism that supports function calling across distinct processes. It supports function calling with arguments passed by value (b) and reference (c).

process and (b) a server which is part of the PALMOS runtime process. The key issue is how the client and server communicate in an efficient manner that eliminates data coping between the different address spaces. We achieve this by using a *Shared Stack* which relies on POSIX shared memory.

5.4.1 Shared Stack

Figure 5.3a shows a function call made by the application to the OpenCL library. The application places the call arguments on the stack and then performs the call. The called function reads the arguments, executes and returns control.

In our scheme, each call to an OpenCL function is forwarded to the PALMOS runtime. The call cannot be performed directly as PALMOS and the application are in different address spaces. Instead there is a *Shared Stack* where the function arguments and return values are stored in a shared memory segment, which is mapped onto the address spaces of both PALMOS and the application. We handle the call as a Remote Process Call (RPC)[15], where the application makes the call request and PALMOS is the remote process that performs it. To illustrate this, see figure 5.3b which shows a function call using the Shared Stack. The application operates in address space 0 and performs an OpenCL call that is served by an OpenCL library that operates in a different address space, address space 1, owned by PALMOS. The application and library share the function call data via the Shared Stack. As shown in section 5.9.3, the use of Shared Stack leads to negligible overhead in comparison to standard OpenCL and other RPC approaches that use system message passing services or MPI[76].

5.4.2 Shared Data

The above approach works if the arguments are passed by value. If arguments are passed by reference[46] and pointers are used, then this approach is not sufficient. Pointers refer to arbitrary addresses of application memory which are not directly accessible to PALMOS. Our *Inter-Space Memory Allocator* (Section 5.5) enables data sharing between the application and PALMOS runtime without data copying. The application uses memory allocated through a custom allocator, which is accessible by the PALMOS runtime.

The RPC OpenCL call is now enhanced as shown in figure 5.3c. Here arg0 of the function call is a pointer referring to an address in the application address space. The memory it points to has been allocated by the Inter-Space Memory Allocator.

PALMOS now can access the application data but the RPC call needs to translate the address value of arg0, which is valid in address space 0 of the application, to a valid address in address space 1 of PALMOS runtime. This is done by the *Address Space Translator* mechanism of the Inter-Space Allocator with negligible overhead.

5.5 Inter-space Memory Allocator

The Inter-Space Memory Allocator enables zero copy data sharing between the applications and PALMOS by operating across multiple address spaces. It also supports data placement control for multi-node NUMA platforms directed by the Resource Manager & Application Scheduler. Although it is used concurrently by multiple processes and threads, its operation remains lock free by design.

5.5.1 Two-Level Memory Allocator

Figure 5.4 shows the structure of our two-level custom allocator [12]. The Central Memory Allocator (CMA) allocates memory chunks from system memory and grants them for use to Local Memory Allocators (LMA). An LMA is instantiated per application thread.

Each chunk is granted exclusively to a single LMA. The LMA manages its chunks as allocation heaps and serves memory allocation requests made by the application. When the allocation heaps of an LMA run out of free memory, the LMA requests a new chunk from CMA. The LMA also yields unused chunks back to CMA. The chunk size is adaptive to the size of memory allocation requests.

The two level design of our allocator guarantees high performance and responsiveness as shown in [12]. Each application thread has its own LMA that manages its own local heaps and its operation is decoupled from other LMAs and the CMA. It contacts CMA rarely, only when new chunks are required.

5.5.2 Address Space Translator

The Inter-Space Memory Allocator uses shared memory segments as chunks. The key issue here is that for each process a shared segment is mapped onto a local address range that is valid only in the address space of the process. However, PALMOS requires access to application data and a mechanism that translates addresses across the different address spaces is required. We introduce the *Address Space Translator* that

converts local addresses to a *global representation* and vice-versa. The global representation is valid across the address spaces and Address Space Translator converts it to valid local addresses for each process.

The global representation has the same size with pointer data types, which is equal to the architecture word size. The most significant half word represents a unique ID given by the CMA, which uniquely identifies a chunk. The least significant half word represents an address offset in the chunk. The translation overhead is negligible, as it only involves simple arithmetic operations. As figure 5.4 shows, address translation takes place in both application and PALMOS.

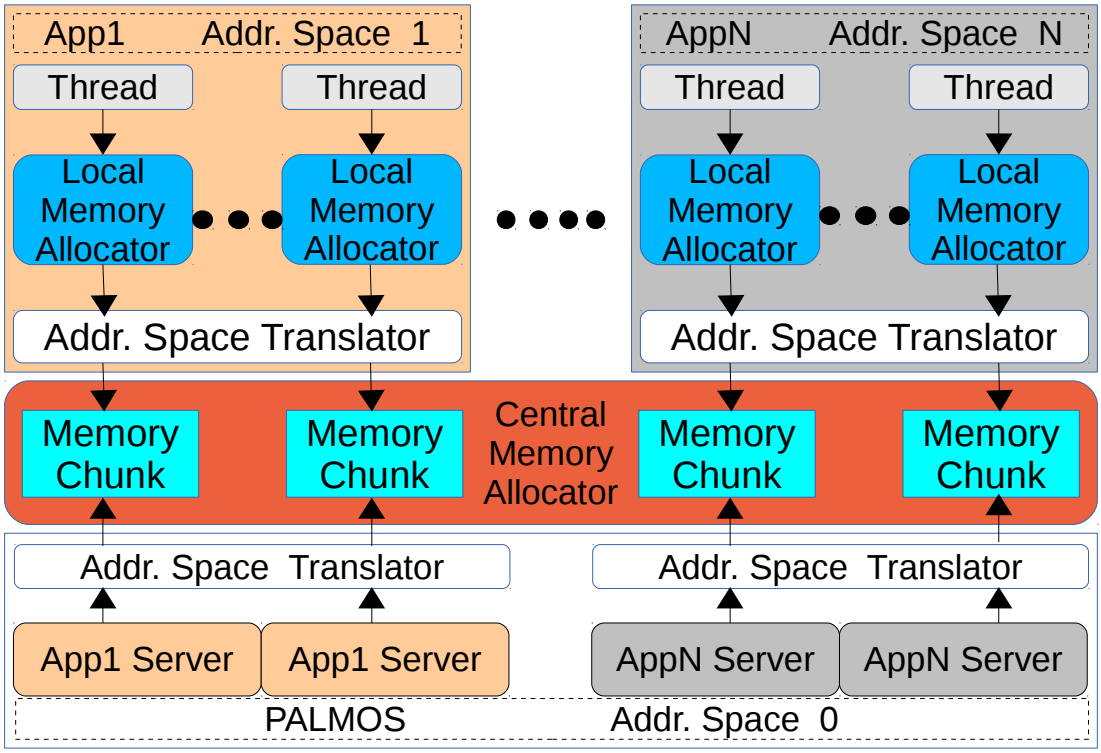


Figure 5.4: Data Sharing with Inter-Space Memory Allocator. Inter-Space memory allocator enables zero copy data sharing between the applications and PALMOS. Each application thread owns a Local Memory Allocator that uses memory chunks, provided by the Central Allocator, as heaps. A special component, called Address Space Translator, is used for the translation of data addresses between the address spaces of applications and PALMOS.

5.5.3 Lock-free Design

Our memory allocator enables data sharing between PALMOS and applications without the need for mutual exclusion as PALMOS and applications interact in a specific manner. An application allocates memory and passes a reference to PALMOS with function calls performed via Virtual OpenCL. While PALMOS processes a call, the application is blocked waiting for the response guaranteeing that the application code neither reads nor updates its data.

Furthermore, PALMOS only reads or updates data in application memory locations specified by the application function calls. It does not allocate new memory that is accessed by the application. An OpenCL implementation may allocate memory internally for its operation but this memory resides only in PALMOS and no sharing is required.

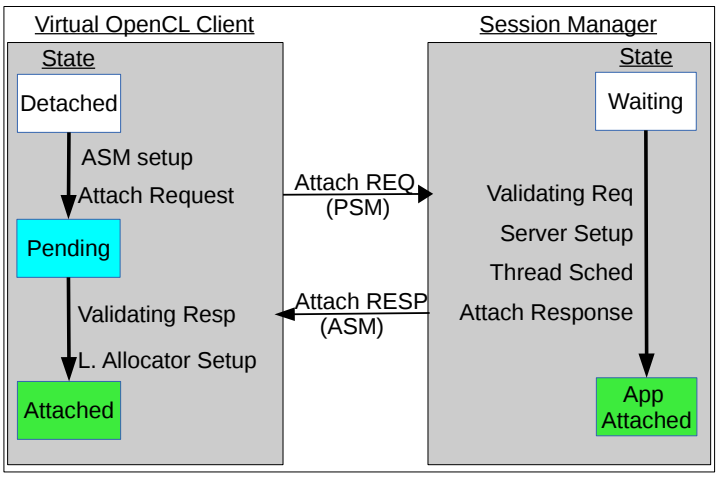
5.6 Resource Manager & Application Scheduler

This section describes the Resource Manager & Application Scheduler (RMAS), the core component of PALMOS. It manages the system resources, monitors application execution and performs efficient scheduling on both host and accelerators. Furthermore, it controls data placement and location aware scheduling for multi-node shared memory systems. RMAS decides which accelerator is used based on its availability. It also reduces the application setup overhead; expensive operations such as accelerator reservation and OpenCL context creation are now done during PALMOS initialization instead of application setup.

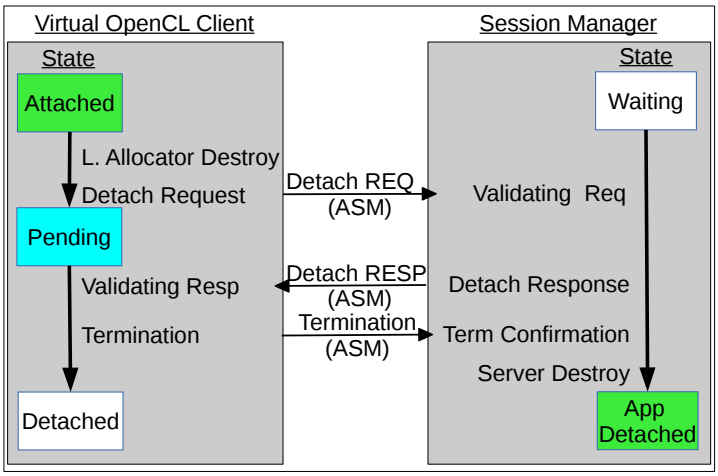
5.6.1 PALMOS Session

PALMOS requires that each application is attached to it via a *session*. This session is used by RMAS to monitor and control the application execution and by our Security component (section 5.7) to build a secure environment for both the application and PALMOS. The session is managed by a protocol with three operations, *Attach*, *Detach* and *Interaction*. Those operations run in the background via the Virtual OpenCL mechanism, no application modification is required or API changes.

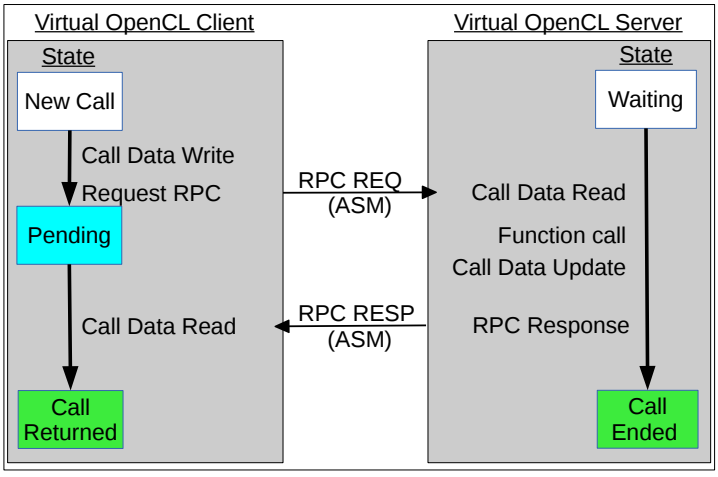
The protocol operations exchange messages between the Virtual OpenCL client of the application and the Virtual OpenCL server of PALMOS runtime via shared memory segments. PALMOS maintains a shared memory segment, named PALMOS Shared



(a) Attach



(b) Detach



(c) Interaction

Figure 5.5: PALMOS Session Protocol. It enables the handshaking of an application with PALMOS and their further interaction. The protocol operations take place in the background via the Virtual OpenCL mechanism and no application modification is required. The design of the protocol is part of our security mechanisms found in section 5.7.

Memory (PSM), where it receives attach requests from Virtual OpenCL clients. The Virtual OpenCL client of an application maintains a shared memory segment, named Application Shared Memory (ASM) which is used as the *Shared Stack* (Section 5.4) and by the attach and detach operations of the PALMOS Session protocol. ASM access is restricted to the application and PALMOS while PSM is accessible by all applications (section 5.7).

Attach: Figure 5.5a shows the Attach operation. PALMOS has created PSM and awaits new requests. The Virtual OpenCL client of the application creates ASM. The client sends an *Attach Request* to PALMOS via PSM. PALMOS receives the request, validates it and instantiates a Virtual OpenCL Server which sends an *Attach Response* back via ASM. The Local Memory Allocator of the application is then instantiated by the Virtual OpenCL client.

Detach: Detach operation, shown in figure 5.5b, takes place just before the application termination. The Virtual OpenCL client destroys the Local Memory Allocator and sends a *Detach Request* to PALMOS. PALMOS validates the request and sends a *Detach Response* back. The client then sends a *Termination* message and application terminates. PALMOS then destroys the corresponding Virtual OpenCL Server. All the communication is done over ASM.

Interaction: Figure 5.5c shows the interaction of application with PALMOS while the session is established. ASM is used as *Shared Stack* for Virtual OpenCL (section 5.4) where the application performs RPC calls to PALMOS.

5.6.2 Application Scheduling

PALMOS has a unified scheduling approach for both CPU and accelerators. It controls the scheduling of host code on CPUs and the scheduling of OpenCL communication and computation operations on accelerators. We deploy a *First In First Out (FIFO)* scheduling policy which follows the same priority semantics with the FIFO scheduling policy of the modern Linux kernel.

Thread Scheduling on Host: RMAS manages multi-threaded applications. Each application thread is attached separately, has its own session and a distinct Virtual OpenCL Server. That choice guarantees high responsiveness. An application thread along with its Server thread are scheduled on the same CPU core to increase data locality.

Operation Scheduling on Accelerators: RMAS manages all the accelerators and schedules operations of host-device communication and kernel computation on them. The application has no direct access to accelerators as it uses Virtual OpenCL, our OpenCL implementation that gives control to PALMOS. RMAS decides which accelerator is used at runtime and performs the actual OpenCL calls.

Operation Multiplexing on Accelerators: A scheduling policy may demand the suspension of an application and the allocation of its computational resources to another application. In the case of host code that runs on CPUs, OS *preemption* takes care of this.

For accelerators, however, the OS has no control over operation ordering or preemption. Accelerators may not support resource sharing and preemption. We solve this by enabling *Operation Multiplexing* that provides operation scheduling control at the granularity of OpenCL operations. OpenCL operations requested by the application are managed by our scheduler which decides which accelerator to use, and the time and order they will be sent to that accelerator. If an application is Paused, its pending operations will not be sent to the accelerator until the application is active again. This is provided by our OpenCL Command Queue implementation.

FIFO Scheduling Policy: When an application starts its execution, a new entry is added in a *Scheduling Priority Queue*. Each entry contains the application state and its priority. We develop a First In First Out (FIFO) scheduling policy with full priority support on both host and accelerators. The scheduling algorithm is driven by two events: the *Attach* and *Detach* of an application.

Attach Event: When a new application gets attached, the scheduler checks if an accelerator is available. If yes, it assigns the accelerator to the application, sets its state to Running and starts its execution. If there is no accelerator available two things may happen. The scheduler picks the running application with the lowest priority. If its priority is lower than the newly attached application, preemption is performed on the host and operation multiplexing on the accelerator. If the priority is higher, the scheduler sets the state of the newly attached application to Waiting and the application waits for its execution.

Detach Event: When an application gets detached, the scheduler sets its state to Completed and releases the accelerator. Then, it checks the *Scheduling Priority Queue* for waiting or paused applications. If there is one, the scheduler sets its state to Running and starts its execution.

5.6.3 NUMA Awareness

PALMOS provides improved scheduling and data placement for multi-node Non Uniform Memory Access (NUMA) platforms. Here the system resources are distributed among multiple nodes. Each node owns part of the system processors/cores and maintains its own physical memory. The nodes communicate and share data through interconnects.

Host Code: The scheduling of application threads and the placement of its data on the same node improves application performance [73]. It reduces the inter-node communication for data sharing or synchronization. Inter-node interconnects typically have lower bandwidth than the node memory.

Accelerator Use: Another factor that affects performance is the location of accelerators. An accelerator is typically attached to a single node and its use by code running on remote nodes requires additional communication through the inter-node interconnects. It is highly preferable for an application to exclusively run and use memory from the node where the accelerator is attached[100][77].

5.7 Security

PALMOS is a user-space virtualization layer for heterogeneous applications that uses shared memory for efficient inter-process communication. While this design choice guarantees high performance, it raises security concerns. Security is key part of our design and we provide a safe environment to both applications and PALMOS.

Secure PALMOS - application interaction: An application interacts with PALMOS through Virtual OpenCL (section 5.4), the Inter-Space Memory Allocator (section 5.5) and the Session Protocol (section 5.6.1) which all use shared memory segments. Our approach for safety is to restrict the access to shared memory segments only to processes that actually should access them. OS provides permission control over shared memory segments in the same manner it does for filesystem access. A shared segment can be read or written only by users and processes that have permissions for those actions. We build a secure environment by enforcing user and group ownership and the corresponding permissions.

Protection of Application Process: Virtual OpenCL forwards computation and communication requests to PALMOS while Session Protocol handles the handshaking between application and PALMOS. Both Virtual OpenCL and Session Protocol access

Application Shared Memory (ASM, section 5.6.1), which is created by the application and needs to be accessed by PALMOS. We set the permission of ASM to be accessible only by the application and PALMOS processes. The use of two shared memory segments in attach operation of Session Protocol enables communication security. PALMOS accepts attach requests through its ”public” shared memory, the PALMOS Shared Memory (PSM) which is accessible to any process. Then, further message exchanging takes place over ASM, which is accessible only by the application and PALMOS. By following that approach we block spoofing attacks and data corruption made by malicious or faulty applications.

Data Privacy: Inter-Space Memory Allocator operates across multiple address spaces and enables data sharing between the application and PALMOS. We ensure data privacy by ensuring that application memory is only accessible by the application and PALMOS. The Inter-Space Memory Allocator uses memory chunks which are allocated as shared memory segments. Each chunk is granted to a Local Memory Allocator (LMA) of an application and needs to be accessed by the application and PALMOS runtime. We set read and write permissions for that chunk only to the processes of the application and PALMOS. Under these permissions the application data is shared safely between the application and PALMOS.

Buffer Overflow & Illegal Memory Access: We avoid Buffer Overflow attacks in Virtual OpenCL by perfoming address range and function type checks in both application and PALMOS. The memory chunks used by Inter-Space Memory Allocator are aligned to page size and have sizes multiples of page size. Any application access outside their address ranges causes immediately application segmentation fault. If there is a misuse, PALMOS terminates the application execution, while it continues serving the other applications.

Special User: For the protection of PALMOS from malicious attacks that can happen through the user environment, the PALMOS process is executed by a special account.

5.8 Experimental Setup

We evaluate PALMOS using a wide range of workloads on 6 platform configurations. Here, we describe how the workloads are selected and present the platform configurations.

5.8.1 Workloads

The workloads considered, consist of between 1 and 64 concurrently running OpenCL programs selected from the Parboil benchmark suite[101] and the largest available dataset is used. Initially, we consider each program in isolation and investigate its performance. Each benchmark was executed 20 times with the average time taken recorded. This is performed in every experiment reported in this chapter to reduce the impact of noise. We then generate multi-program workloads by selecting multiple benchmarks from Parboil. We looked at workload groups containing 1, 4, 16, 32 and 64 programs to investigate scaling. We randomly selected 50 distinct combinations of benchmarks for each workload group and report the gmean execution time.

5.8.2 Platform

Our evaluation platform is a NUMA x86 system with two nodes. Each node has an Intel Xeon E5-2620 CPU running at 2.00GHz and 8 Gigabytes of DDR3 memory running at 1333MHz. The system bus is a QPI interconnect. We have attached two NVIDIA and one AMD GPUs. One NVIDIA and one AMD GPUs are attached on the first node through its local PCI-Express interface. The second NVIDIA GPU is attached on the second node through its local interface. Both NVIDIA GPUs are Tesla K20c[80] and the AMD GPU is Tahiti 7970[3]. The Operating System is Linux with kernel version 3.7 and three OpenCL implementations, Intel OpenCL 1.2 (Build 67279), NVIDIA OpenCL 1.1 (CUDA 6.0.1) and AMD OpenCL 1.2 (Build 1348.5). Both CPUs are treated as a single accelerator.

We define distinct platform configurations by making different number of accelerators available for use. Our configurations are (a) *Single NVIDIA GPU*, (b) *Single AMD GPU*, (c) *Intel CPU*, (d) *Two NVIDIA GPUs*, (e) *Two NVIDIA and one AMD GPUs*, (f) *One NVIDIA GPU and Intel CPU*, (g) *Two NVIDIA GPUs and Intel CPU*, (h) *Two NVIDIA GPUs, one AMD GPU and Intel CPU*.

Three benchmarks, `histo`, `mri-gridding` and `sad`, fail to run if the CPU is used as the accelerator. In case of AMD GPU, `mri-gridding` fails too. The PALMOS scheduler handles this situation. The benchmarks that fail with one accelerator type are not scheduled on it. These benchmarks are not fully compatible with OpenCL standard.

5.8.3 Comparison to existing approaches

Vendor Solutions The NVIDIA Kepler architecture enables multi-program execution and supports concurrent kernel executions that overlap with host-accelerator communication. We have modified our FIFO scheduling policy to enable concurrent execution of multiple programs on NVIDIA GPUs. There is, however, limited space for performance improvement because benchmarks tend to consume all the GPU resources leaving no space for concurrent execution[85].

Message Passing Based Solutions There have been a number of message passing based schemes that allow sharing of accelerators. As these require extensive development and modification of the OpenCL benchmarks to use an extended API, our evaluation is more limited. We implemented the best performing scheme described in [11] and evaluated a small number of task scenarios. We compare PALMOS against it on our shared memory platform.

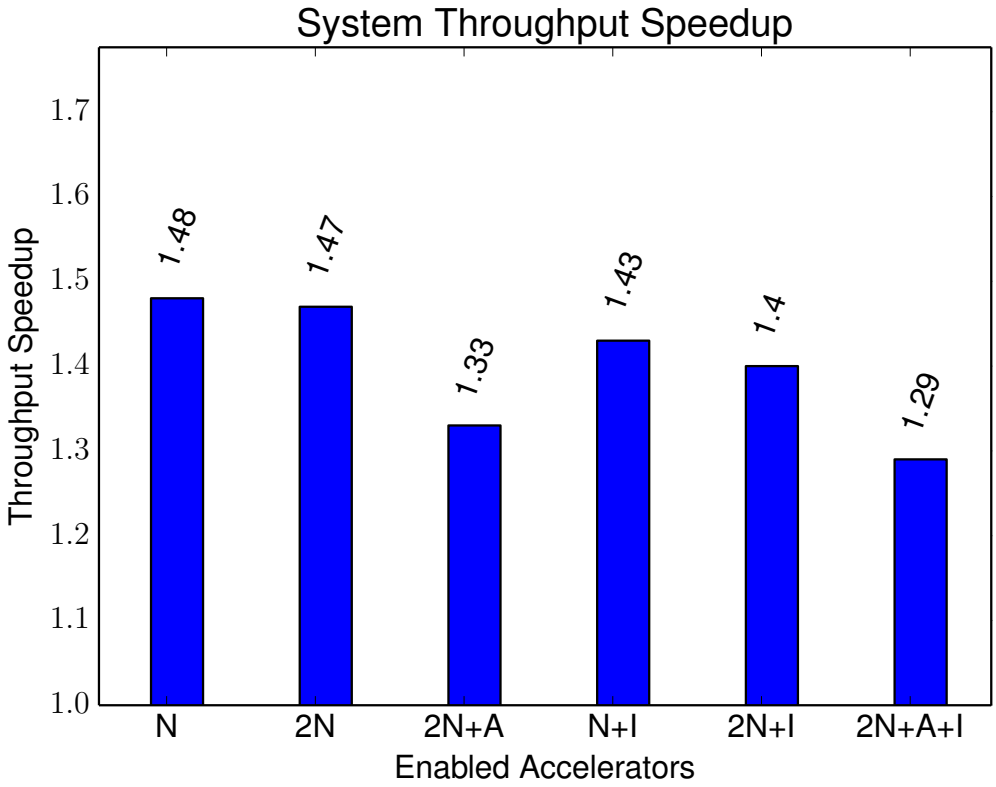


Figure 5.6: Representative of throughput speedup results delived by PALMOS for multi-program workloads of 64 benchmark instances. We consider the following 6 platform configurations: **N**: 1 NVIDIA GPU, **2N**: 2 NVIDIA GPUs, **2N+A**: 2 NVIDIA and 1 AMD GPUS, **2N+I**: 2 NVIDIA GPUS and Intel CPU, **2N+A+I**: 2 NVIDIA and 1 AMD GPUs and Intel CPU.

5.9 Results

The main goal of our work is improving multi-program system throughput. A summary of the results is shown in figure 5.6 for multi-program workloads consisting of 64 program on each of the 6 platform configurations. Overall, PALMOS delivers significant speedups ranging from from 1.29x to 1.48x. The remainder of this section examines in greater detail the performance of our approach. Section 5.9.1 examines individual benchmark performance improvement. Section 5.9.2 then evaluates performance for varying sized multi-program workloads on each platform configuration. Section 5.9.3 provides a direct performance comparison against the Standard OpenCL environment and an alternate, state-of-the-art message passing based approach.

5.9.1 Single application performance

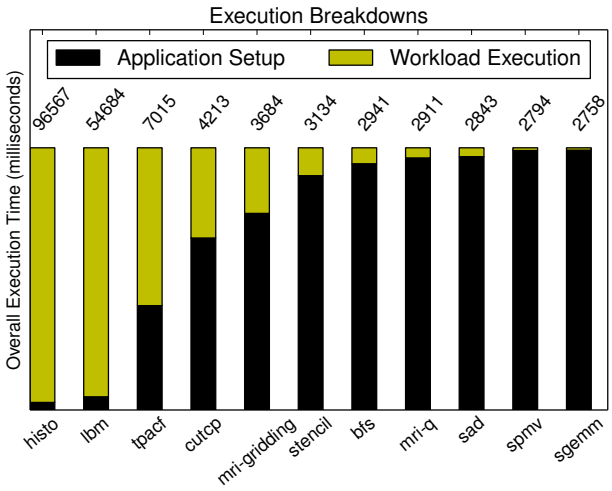
Here, we evaluate the performance of each application running in isolation. We examine (a) reduction of application setup overhead and (b) NUMA aware scheduling.

5.9.1.1 Reduced Setup Overhead

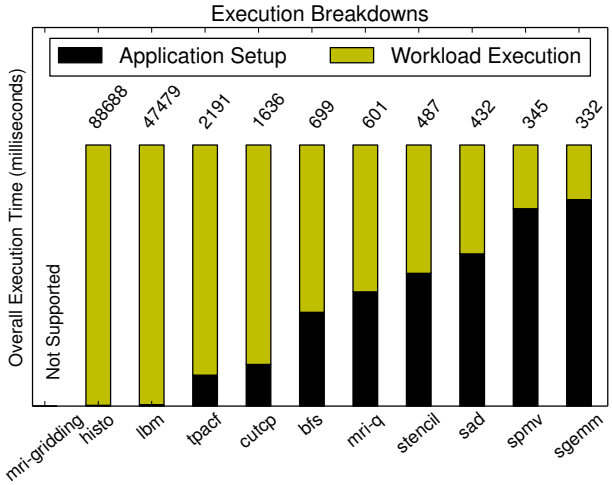
In standard OpenCL, an application has significant setup overhead as it must reserve an accelerator and create an OpenCL context. In the PALMOS environment, those operations happen once at initialization and overhead is significantly reduced.

NVIDIA GPU: Figure 5.7a shows the total execution times of benchmarks with an NVIDIA GPU as accelerator and the percentage time spent in setup and execution. Benchmarks, such as `histo` and `lbm` spend as little as 3% and 5% of their execution on setup, which are reduced by PALMOS to 0.2% and 0.3% as shown in table 5.1, rows 1 and 2. In contrast, benchmarks such as `spmv` and `sgemm` spend 97% and 97.6% of their execution on setup, values that are significantly reduced to 12.2% and 12.2% (table 1, rows 10 and 11).

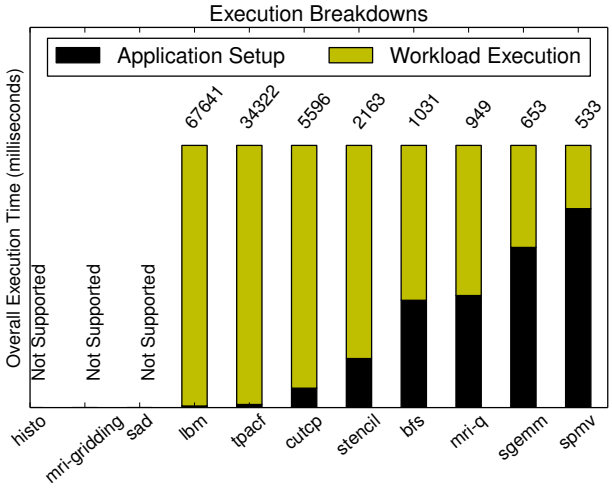
AMD GPU: Figure 5.7b shows execution time breakdown for the AMD GPU. While AMD kernel execution times are approximately the same as NVIDIA, setup overhead is approximately an order of magnitude faster, 328ms compared to 2630ms. Benchmarks `histo` and `lbm` now spend 0.2% and 0.5% of their execution on setup, values that are reduced by PALMOS to 0.1% and 0.2% as shown in table 5.1. Benchmarks such as `spmv` and `sgemm` spend the 75.2% and 79.6% of their execution on setup, those values get reduced to 8.6% and 8.9%.



(a) NVIDIA GPU as Accelerator



(b) AMD GPU as Accelerator



(c) CPU as Accelerator

Figure 5.7: Normalized Execution Time Breakdowns of benchmarks on Standard OpenCL environment. The stacks present the time spent on application setup and on actual workload execution. Setup times can be prohibitive regardless the accelerator vendor. PALMOS design reduces this overhead dramatically.

Bench	N(%)	NP(%)	A(%)	AP(%)	I(%)	IP(%)
histo	3	0.2	0.2	0.1	N/A	N/A
lbm	5	0.3	0.5	0.2	0.6	0.2
tpacf	39.1	5	11	2	1.1	0.7
cutcp	65	7	16	2.4	7	1.5
mri-gr	74.6	8.5	N/A	N/A	N/A	N/A
stencil	89	10.1	50	8.3	18	4.7
bfs	93.2	11.4	35	7.0	40	7.9
mri-q	95.7	12.1	43	7.6	42	7
sad	96.5	12.2	58	8.7	N/A	N/A
spvm	97	12.2	75.2	8.6	76.2	8.6
sgemm	97.6	12.2	79.6	8.9	61	8.2

Table 5.1: Comparison of Setup Overhead on Standard OpenCL and PALMOS environment. The results present the proportion (%) of total execution time spent on setup per benchmark. **N**: NVIDIA GPU, **NP**: NVIDIA GPU with PALMOS, **A**: AMD GPU, **AP**: AMD GPU with PALMOS, **I**: Intel CPU, **IP**: Intel CPU with PALMOS.

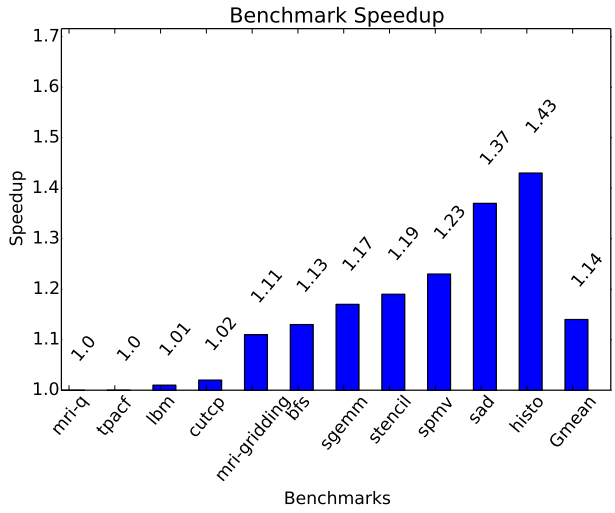
Intel CPU: Figure 5.7c shows the total execution times when the CPUs are used as a single accelerator. The executions of OpenCL kernels on CPUs are slower and Intel OpenCL has lower setup times. Benchmarks such as `lbm` and `tpacf` spend 0.6% and 1.1% of their execution on setup, those values get reduced by PALMOS to 0.2% and 0.7%, as shown in table 5.1. Benchmarks such as `sgemm` and `spmv` spend 61% and 76.2% of their execution on setup and PALMOS reduces it to 8.2% and 8.6%.

Benchmark Classification: Based on the proportion of the total execution time benchmarks spend on setup we classify them in two groups for later evaluation. The first includes the benchmarks with **High** workload that have low setup overhead, which are `histo`, `lbm`, `tpacf`, `cutcp` and `mri-gridding`. The second group includes benchmarks with **Low** workloads that have high setup overhead, which are `stencil`, `bfs`, `mri-q`, `sad`, `spmv` and `sgemm`.

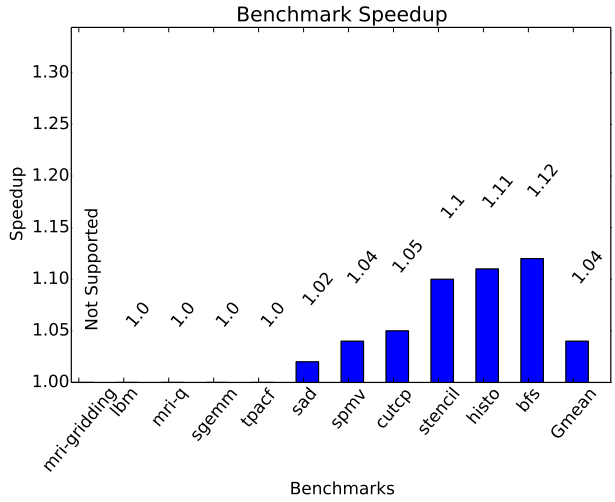
5.9.1.2 NUMA Aware Scheduling and Data Placement

Here, we report the additional speedups delivered by the enhanced NUMA aware mode of PALMOS. This applies only when GPUs are used as accelerators because both CPUs are treated as a single accelerator.

NVIDIA GPU: Figure 5.8a shows the speedups for NVIDIA GPUs when PALMOS is NUMA aware which range from 1.0x to 1.43x with a geometric average of 1.14x. Benchmarks that perform significant host-accelerator communication such as



(a) NVIDIA GPU

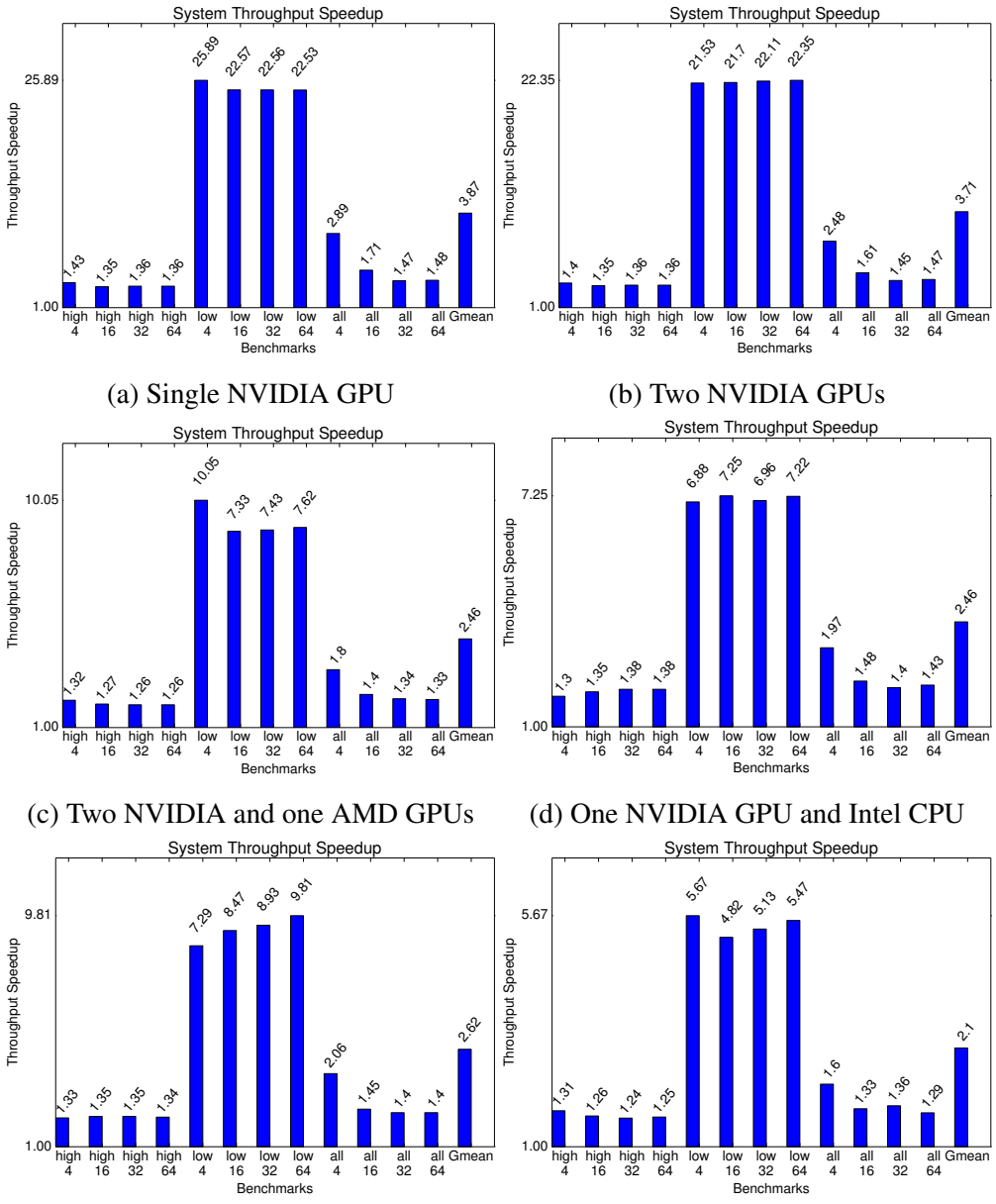


(b) AMD GPU

Figure 5.8: Speedup (NUMA aware scheduling and Data Placement). Execution speedups for individual benchmarks delivered by the NUMA mode of PALMOS. The baseline is the standard PALMOS environment. Logarithmic scale. GPU only.

`sad` and `histo` present notably higher speedups than benchmarks with reduced communication such as `tpacf` and `cutcp`.

AMD GPU: Figure 5.8b shows the speedups for AMD GPU. The reported speedup values are now lower and range from 1.0x to 1.12x. However, benchmarks with significant host-accelerator communication, such as `histo` and `bfs` still present the highest speedups of 1.11x and 1.12x.



(a) Single NVIDIA GPU

(b) Two NVIDIA GPUs

(c) Two NVIDIA and one AMD GPUs

(d) One NVIDIA GPU and Intel CPU

(e) Two NVIDIA GPUs and Intel CPU

(f) Two NVIDIA and one AMD GPUs, Intel CPU

Figure 5.9: Throughput Speedup Results delivered by PALMOS. Multi-program sets are executed on 6 platform configurations where different number and types of accelerators are available. We consider 3 groups of benchmarks, the **High** Workload, **Low** Workload and **All** benchmarks. Group size varies from 4 to 64 benchmark instances. The baseline is the Standard OpenCL environment. Logarithmic Scale.

5.9.2 Multi-program performance

This section evaluates multi-user, multi-program performance in more detail. We consider three benchmark groups: **High**, **Low** and finally **All** which contains all the benchmarks of the previous two groups.

To evaluate multi-tasking, we randomly generate sets of benchmarks that run concurrently on the system. We consider sets of 4, 16, 32 and 64 benchmark instances. For each benchmark group and set size we randomly generate 50 permutations of the benchmarks belonging to that group. We run each of these permutations 20 times to eliminate noise. We present a geomean throughput speedup for each benchmark group and set size as shown in figure 5.9.

Single NVIDA GPU: Figure 5.9a shows the throughput speedup when a single NVIDIA GPU is available. Here, for **High** workloads as the size of the workload increases from 4 to 64, the improvement gains level to around a 1.36x speedup as resource contention begins to increase. For **Low** workloadsour improvements are actually more significant, ranging from 25.89x to 22.53x. The decrease happens due to resource contention. Combining all benchmarks together and running them gives improvements from 1.48x to 2.89x. The **High** workload sets, that have long execution times, have lower performance gains than **Low** workload sets. However, for the **All** workload sets, PALMOS provides significant throughput speedups. Averaged across each scenario we see a significant 3.87x improvement over standard OpenCL.

Two NVIDIA GPUs: Figure 5.9b gives the throughput speedup results when two NVIDIA GPUs are available. When compared to figure 5.9a, the speedups obtained are highly consistent. However, the results for **Low** workloads are now reduced and they range from 21.53x to 22.35x. The results for **All** workloads are slightly reduced ranging from 2.48 to 1.47. This happens because (a) the same workloads are now computed on two accelerators and the concurrency reduces the potential performance improvements, (b) some overhead is introduced for using two accelerators. The NUMA aware scheduling reduces that overhead but still many driver and OS operations are centralized. Overall, the average improvement across all the scenarios is 3.71x.

Two NVIDIA and one AMD GPUs: Figure 5.9c shows the results for two NVIDIA and one AMD GPUs. Here the results follow the trends of the previous configurations but the throughput speedup gains are reduced giving a geomean improvement of 2.46 while **Low** workloads report significant reductions ranging from 10.05 to 7.62x. The reason for the relative reduction is (a) the lower setup cost introduced by AMD

OpenCL, (b) the higher overhead of managing and using three accelerators of different vendors and (c) the fact that the same workloads are now computed by three accelerators concurrently. The **Low** workload set of 4 benchmarks reports a higher throughput speedup of 10.05x in comparison to larger **Low** sets because of lower resource contention.

One NVIDIA GPU and Intel CPU: Figure 5.9d shows the results when one NVIDIA GPU and Intel CPU are available as accelerators. The throughput speedup trends remain the same with the previous configurations but they are now degraded, specially for **Low** workloads which now have throughput speedup ranging from 6.88 to 7.22. The improvement rates are degraded mainly because of (a) the longer execution times on CPU and (b) the reduced setup overhead of Intel OpenCL. The geomean speedup is 2.46x.

Two NVIDIA GPUs and Intel CPU: Figure 5.9e shows the throughput speedup results given by PALMOS when two NVIDIA GPUs and Intel CPU are available. The throughput speedup results are now higher in comparison with the `One NVIDIA GPU and Intel CPU` configuration. The reason is that there are two NVIDIA GPUs in use (a) that lead to faster executions and (b) have higher setup overheads in contrast to Intel CPU. The geomean improvement is 2.62x.

Two NVIDIA GPUs, one AMD GPU and Intel CPU: Here we make all the accelerators available. The results, shown in figure 5.9f, follow the same trends as before but the throughput values are further reduced. **Low** workloads now range from 5.67x to 5.47x and the mixed, **All** workloads from 1.6 to 1.29. **High** workloads remain on the same levels. The reason is (a) the overhead of using multiple accelerators of three different vendors, (b) the lower setup overhead of Intel and AMD OpenCL implementations, (c) the slower executions on CPU and (d) the fact that the same workloads are executed concurrently on 4 accelerators. However the geomean throughput speedup, 2.1x, is still significant.

5.9.3 PALMOS against existing approaches

In this section we evaluate the performance of PALMOS against the default OpenCL environment and an alternative approach [11] which uses message passing for communication and we refer to it as **Cloud**. We consider four versions of **Cloud**. The first version controls all the accelerators centrally, where there is little application setup overhead. The second, is a secure version called **secCloud**, where a new OpenCL

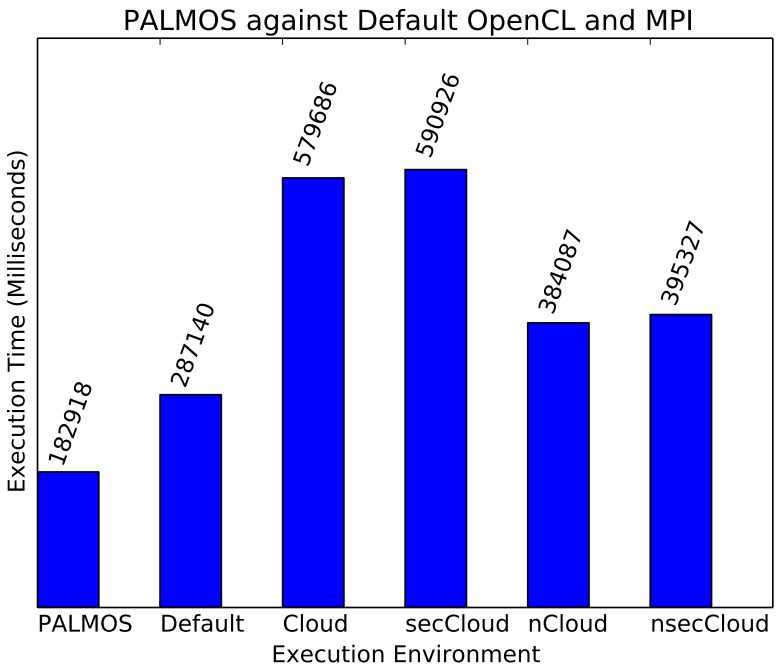


Figure 5.10: This figure shows the absolute times for the execution of (`spmv`, `histo`, `spmv`, `histo`) set on PALMOS, Default OpenCL and the 4 versions of Cloud infrastructure.

context is created per application The last two versions are NUMA aware versions of Cloud and secCloud, called **nCloud** and **nsecCloud**. As the OpenCL benchmarks have to be modified to use **Cloud**, we perform a smaller evaluation than the previous section. Here, we use one of the NVIDIA GPUs as the accelerator.

Performance Overview: Here, we take a pair of Parboil benchmarks, `histo` from the **High** workload group and `spmv` from the **Low** workload group and we create a workload set, (`spmv`, `histo`, `spmv`, `histo`). We run it on PALMOS, Standard OpenCL, and, once suitably modified, on the four Cloud infrastructures and the results are shown in figure 5.10. PALMOS outperforms all the other approaches. PALMOS is 57% faster than default OpenCL execution, where the speedup comes from the reduced setup overhead and NUMA awareness. It outperforms Cloud and secCloud by 217% and 223% respectively. The Cloud approaches require extensive data copies across different address spaces, something that PALMOS avoids by using shared memory. Cloud is 7% faster than its secure version, secCloud. The NUMA aware versions, nCloud and nsecCloud are significantly faster. The data placement improves the data copying performance. However, both remain significantly slower.

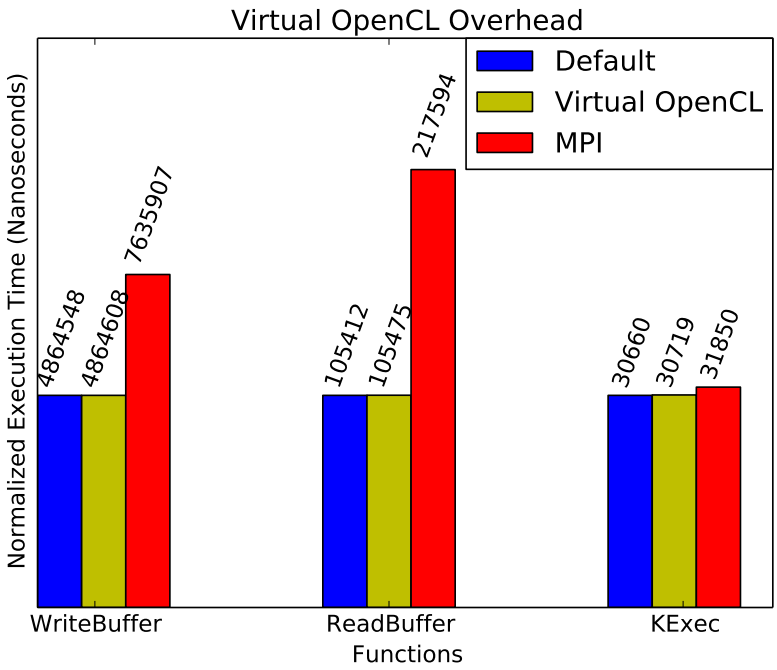


Figure 5.11: Virtual OpenCL overhead. Comparison against default call convention and RPC via MPI. Results are normalized to Default.

Virtual OpenCL Performance Analysis: Figure 5.11 shows the execution times for function calls performed with (a) the default call convention of standard OpenCL, (b) Virtual OpenCL and (c) MPI when executing **spmv**.

Kernel execution, **clEnqueueNDRangeKernel**, as expected, is roughly the same for each scheme. In the case of host-accelerator communication, **clEnqueueWriteBuffer** and **clEnqueueReadBuffer**, Virtual OpenCL introduces negligible overhead. MPI however introduces an overhead of 1.56x and 2.06x as data need to be copied before the actual host-accelerator communication can take place. These two calls consume significant proportion of the application execution and lead to critical execution slowdowns for MPI/message passing based schemes.

5.10 Summary

In this chapter, we have presented PALMOS, a secure user-space virtualization layer that integratees accelerator resources. It targets heterogeneous systems found in data center nodes and requires mo modification of OS, OpenCL or the application. Our approach reduces application setup overhead, enables inter-vendor accelerator sharing and provides efficient platform aware scheduling. Our evaluation on multiple platforms

configurations with workloads ranging from 1 up to 64 applications shows average improvement from 1.29x to 3.87x throughput speedup. We also show that our approach outperforms both vendor accelerator sharing facilities and message passing solutions.

The next chapter presents a technique and software stack component that enable resource sharing control on accelerators, while supporting software managed scheduling on accelerators. This work remains transparent to existing systems and applications and requires no modifications or recompilation.

Chapter 6

Resource Sharing Control on Accelerators

This chapter presents a runtime and Just In Time compiler infrastructure that enable resource sharing control on accelerators, while supporting software managed scheduling on accelerators. Our infrastructure remains transparent to existing systems and applications and requires no modifications or recompilation. We evaluate on NVIDIA and AMD GPU platforms. We enforce fairness in accelerator resource sharing and deliver fairness improvements ranging from 6.8x to 13.66x for different workloads. Furthermore, we deliver system throughput speedups ranging from 1.13x to 1.31x.

The remainder of the chapter is organized as follows. The challenges and contributions of this chapter are given in section 6.1. Section 6.2 describes our motivation for accelOS. Section 6.3 introduces our resource sharing scheme for accelerators. Section 6.4 presents a high level overview of accelOS infrastructure. Sections 6.5 and 6.6 present the key components and features of accelOS. We present our experimental setup and evaluation in sections 6.7 and 6.8. Section 6.9 discusses chapter summary.

6.1 Introduction

Accelerators, such as Graphic Processing Units (GPUs) and Field-Programmable Gate Arrays (FPGAs), are key components of modern parallel platforms. They deliver high computational throughput with reduced power for data parallel applications. However, this raw hardware performance comes at a software cost. Although highly parallel, the accelerators are managed as co-processors and support a limited number of concurrent

kernel executions at a time.

While sharing of accelerator resources is not an issue for dedicated application systems found in HPC, it is a real barrier for accelerator adoption in general purpose servers and data centers. Such systems typically host diverse parallel applications which cannot efficiently share and access accelerators with the existing software stack. There is no fair resource sharing on accelerators for execution requests arriving concurrently from distinct applications. This lack of control makes it impossible to provide quality of service guarantees and directly affects the overall fairness of the system.

Modern computing systems need a mechanism that allows accelerators to be shared fairly among several concurrent kernel executions. This should incur minimal overhead and ideally support immediate deployment in existing systems with minimal disruption.

This chapter develops a portable and transparent approach for accelerator sharing control. It enables concurrent space sharing of the accelerator by multiple kernels without any change to the application code, the Operating System or GPU hardware. It can be used immediately on existing hardware and OpenCL software stacks [58].

We achieve this by deploying a host runtime environment and a Just In Time(JIT) compiler. The runtime determines the amount of resources required for the execution of a single work group[1] of every kernel. The runtime then uses this information to narrow down the work groups of every kernel execution in order to control the resources it reserves on the accelerator. To guarantee the correctness of the application we need to perform the computation of the original number of work groups. This is done by dynamically assigning additional work to the newly reduced number of work groups. This procedure requires the transformation of kernel codes and is performed transparently by our JIT compiler. This scheme ensures all kernels can run concurrently, have equal resources, benefit from dynamic work group scheduling and no user or system code has to be modified.

The need for concurrent space sharing of GPUs is well known; in fact GPU manufacturers have separate hardware queues specifically for this purpose. These are intended to allow efficient utilisation by different application streams and kernels. The NVIDIA architecture is a good example. In practice, however, although 2 or more kernels can be sent for execution, the hardware scheduler currently assigns all resources to which ever one arrives first. There is no notion of fair access.

[1] A work group, in OpenCL, represents a subset of kernel execution threads (work items) that needs to be scheduled on a single compute unit of the accelerator.

There have been hardware based proposals to improve performance [35][79][20][16] and memory bandwidth[89]. They do not, however, investigate multi-kernel scheduling and fair resource sharing. Furthermore, they crucially require hardware modifications that are not currently available.

There has also been significant interest in software approaches to GPU sharing for performance [85][1][43] and power efficiency[52]. However, these techniques require static code merging with no dynamic control and do not investigate fair resource sharing. Furthermore, they raise security concerns because they merge kernel codes of different applications and users. There is also significant work proposing host runtime and Operating System techniques for managing accelerator resources[56] [56][91][71]. However, they focus on allowing tasks to be easily allocated to a CPU or GPU, rather than resource sharing control on accelerators.

Prior work has extensively investigated system resource sharing for non accelerator based systems [98][17][104][114][34] and a number of metrics have been proposed for quantifying fairness[41][49][36]. However, there has been limited work on fair sharing of accelerator resources, a problem we directly address in this chapter. We adopt the fairness metric proposed in [32] and extend it to quantify efficiently fairness on accelerators. We use their definition of fairness where the slowdowns of equal-priority applications running simultaneously on the accelerators are the same.

This work presents accelOS, a software stack component that consists of a runtime and a Just In Time compiler and enables resource sharing control and software managed scheduling on accelerators. It integrates with the existing software stack; its operation remains transparent to the application, OS and runtime libraries and there is no requirement for code modifications or recompilation. accelOS operates on the top of existing hardware facilities and their operations are orthogonal.

To summarize, this chapter makes the following contributions:

- A runtime and JIT compiler infrastructure that enables resource sharing control on accelerators.
- Fair accelerator sharing for multi-kernel executions.
- Dynamic, software managed scheduling on accelerators.
- Seamless integration with existing systems and portability across accelerator vendors, OSes and applications.

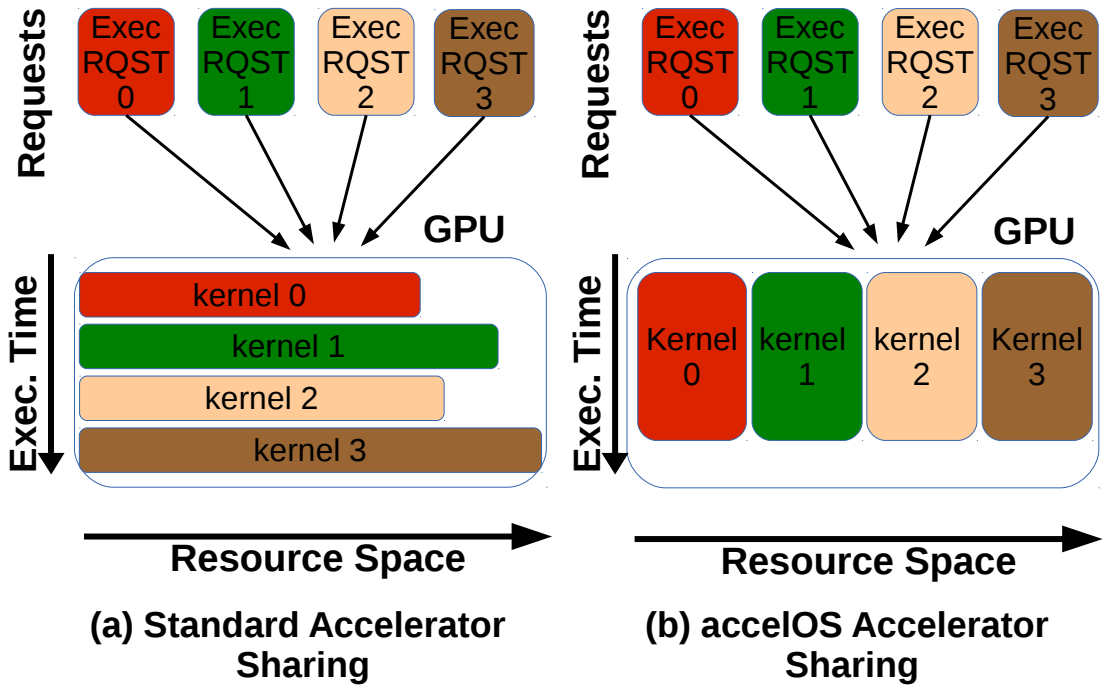


Figure 6.1: Accelerator sharing on standard OpenCL (a) against accelOS (b). Standard OpenCL does not provide resource sharing control leading to unfair sharing and poor concurrent kernel executions. In contrast, accelOS enforces fair accelerator sharing leading to efficient concurrent kernel executions and throughput improvements.

Our approach is evaluated extensively by using workloads consisting of multiple OpenCL kernels from the Parboil benchmark suite. We first evaluate all pairwise combinations of kernels ($25 \times 25 = 625$ in total). We then evaluate 16384, 4-kernel combinations randomly selected from the 390265 combinations and 32768 8-kernels randomly selected from the 1.5×10^{11} combinations We evaluate performance in terms of fairness, processor sharing, system throughput and system overhead. We compare our approach to the hardware baseline and the *Elastic Kernels* approach [85]. To show accelOS portability, we evaluate its performance on two modern heterogeneous platforms from two different manufacturers.

We dramatically improve fairness, ranging from 6.8x to 13.66x. This has the added bonus of improving system throughput on average from 1.13x to 1.31x. Our scheme incurs no overhead due to our compiler optimizations, in fact we actually improve isolated kernel execution times due to dynamic scheduling. We deliver an average execution speedup of 1.07x and 1.1x on NVIDIA and AMD GPUs.

6.2 Motivation

Consider figure 6.1 which graphically presents the execution of four kernels arriving from distinct applications when concurrently requesting execution on a modern dedicated GPU such as NVIDIA's Tesla k20m or AMD's R9 295X2.

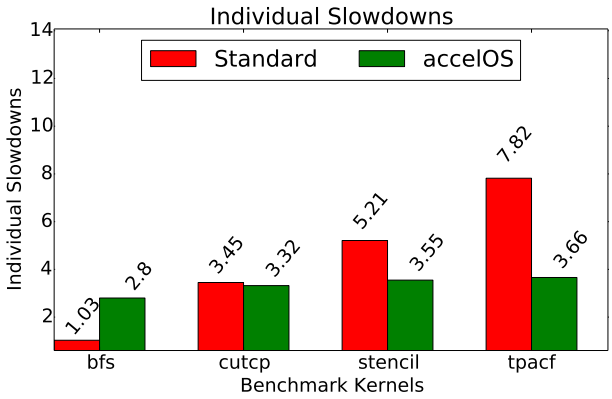
Figure 6.1a illustrates typical system behavior when the standard software stack is used. Rather than executing concurrently, each kernel is executed sequentially in turn. The reason is that each kernel is able to use the majority of the system resources, leaving little space to execute the others and leads to sequential executions. The existing software stack does not support resource sharing control on the accelerator and the accelerator architecture does not support preemption. This leads to unfair accelerator sharing for different applications and their users.

Figure 6.1b presents the system behavior where accelOS infrastructure is in place. accelOs confines resource allocation for kernels so that they have more work per thread but less concurrent threads and thus demand less system resources. This is done dynamically by altering the number of work groups for kernel executions. It also requires software managed scheduling of the original work groups on accelerators and we support it. As can be seen the accelerator resources are now allocated equally among the four kernel executions. This new behavior leads to fair accelerator sharing, concurrent kernel executions and improved throughput.

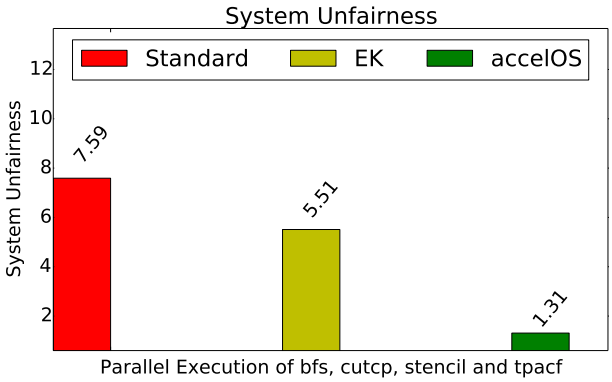
6.2.1 Motivational Example

To make this concrete, we consider the performance of 4 kernels, *bfs*, *cutcp*, *stencil*, and *tpacf* when concurrently presented for execution using the standard software stack and accelOS on NVIDIA platform. Figure 6.2a shows the slowdown of each kernel when executed concurrently relative to executing in isolation. The standard scheme executes them sequentially and as expected bfs has the least slowdown as it is executed first while program tpacf has the largest slowdown as it is executed last. accelOS slows each kernel more evenly giving fairer access to the GPU. Using the unfairness metric [32], this means that accelOS is 5.79x times fairer as shown in figure 6.2b. As accelOS is better able to use system resources, it actually improves system throughput as well, 1.31x over the standard scheme, as shown in figure 6.2c.

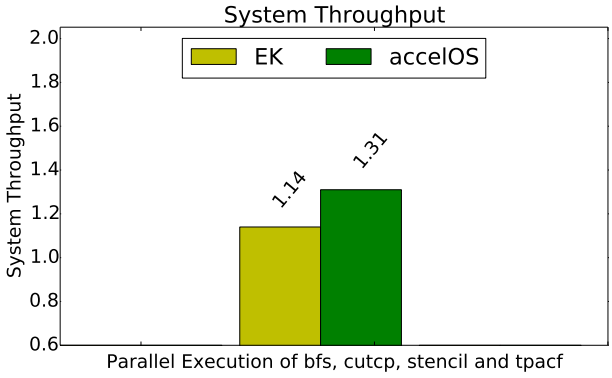
An alternative scheme, *Elastic Kernels* [85], attempts to statically merge kernels when system resources may not be fully utilised. This scheme is able to improve



(a) Individual Slowdowns. accelOS delivers reduced and balanced slowdowns for the kernel executions. This denotes fair accelerator sharing and concurrent kernel executions.



(b) System Unfairness. We compare the system unfairness delived by standard OpenCL and *Elastic Kernels (EK)* against accelOS. Lower is better. accelOS outperforms the other approaches.



(c) System Throughput Speedup. We compare against the *Elastic Kernels (EK)*. The baseline is the standard OpenCL. Higher is better. accelOS delivers better throughput performance.

Figure 6.2: Motivational Example for accelOS. Parallel execution of *bfs*, *cutcp*, *stencil*, and *tpacf*. accelOS outperforms other approaches in system fairness and throughput results.

system throughput by 1.14x but does not improve fairness as it does not allocate resources evenly. As can be seen in figures 6.2b and 6.2c accelOS delivers better system throughput and fairness results than Elastic Kernels.

6.2.2 Standard Scheduling Approach

OpenCL and CUDA do not expose any control on how accelerator resources are allocated among concurrent kernel execution requests. In practice, the execution request that *arrives* first tends to reserve all the available resources. This happens for two reasons. First, the hardware and firmware of the accelerator do not constrain the resources a kernel execution uses. Second, a kernel execution request typically represents a computational range that is large enough to occupy all the accelerator resources.

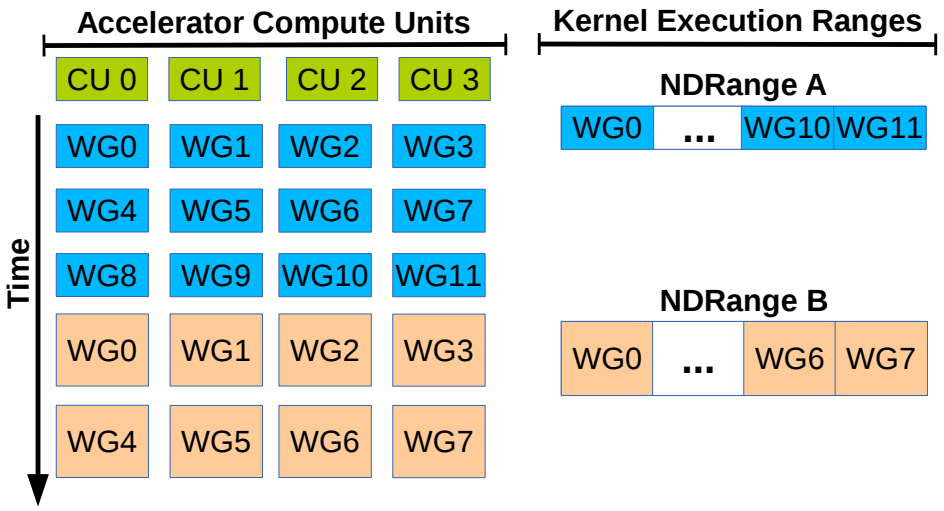
Consider figure 6.3 which illustrates accelerator sharing and work group scheduling for two parallel kernel execution requests. Here, the accelerator has four compute units (CUs) and two kernels, A and B. A's Kernel Execution Range (NDRange) consists of 12 work groups (WGs), while B has 8 work groups.

Figure 6.3a illustrates accelerator sharing and work group scheduling as it happens today on modern accelerators. Here, the hardware/firmware scheduler assigns work groups to compute units based on static heuristics. There is no control on how work groups are assigned to the compute units. Under this scheme the kernel that arrives first, kernel A in this example, allocates resources across all the compute units and the scheduler assigns work groups across the units in a round robin fashion. The work groups of kernel B start executing only after the completion of kernel A. The lack of resource sharing control leads to serialized kernel executions and unfair sharing.

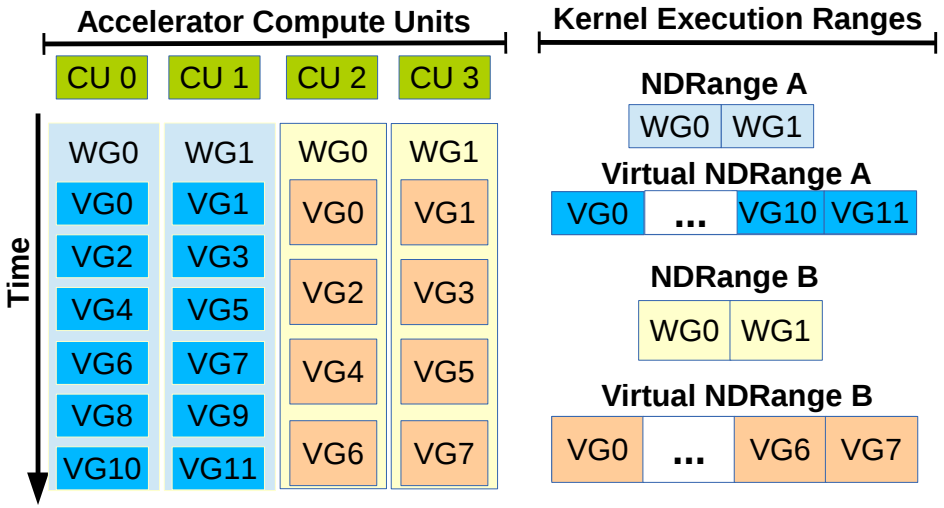
6.2.3 accelOS: Software Scheduling & Resource Sharing Control

Figure 6.3b shows our approach. The number of work groups for both kernel executions is now reduced. In our example both A and B now have just 2 work groups. The work groups of kernel A start executing on compute units 0 and 1, while the work groups of kernel B executing on compute units 2 and 3.

To ensure the original computation is performed, we have to compute all the original work groups of each kernel execution. First of all, each of the original 12 work groups of A (8 of B) are stored in a software queue and we refer to them as *virtual groups*. The software queue is stored in accelerator memory and we refer to it as *Virtual NDRange A (B)*. Next, our JIT compiler transparently modifies the kernel code



(a) In standard OpenCL, there is no resource sharing control. The work groups (WGs) of every Kernel Execution Range (NDRange) are assigned for computation to the accelerator compute units (CUs). This is done by static schedulers in a round-robin fashion and there is no control on how the work groups are assigned to the compute units. The kernel execution that arrives first allocates all the accelerator resources. When the work groups of the first kernel have been computed, work groups of the second kernel start executing on the compute units. The lack of resource sharing control leads to serialized kernel executions and unfair accelerator sharing.



(b) accelOS enables resource sharing control. This is done by altering the number of work groups (WGs) of Kernel Execution Ranges. We narrow down the number of work groups to control the number of compute units (CUs) a kernel uses at a time. This way we allow multiple kernels to perform concurrently on the accelerator and achieve space sharing. We preserve the original Execution Range at software level. We store it in accelerator global memory and we now name it *Virtual NDRange* and its work groups, *Virtual Groups* (VGs). Our JIT compiler transforms the kernel code. The new kernel code performs software work group scheduling by accessing Virtual NDRange, retrieving and computing virtual groups at runtime.

Figure 6.3: Example of accelerator sharing for two kernel execution requests. We compare standard OpenCL (a) against accelOS (b). accelOS supports resource sharing control which enables fair accelerator sharing and concurrent kernel executions.

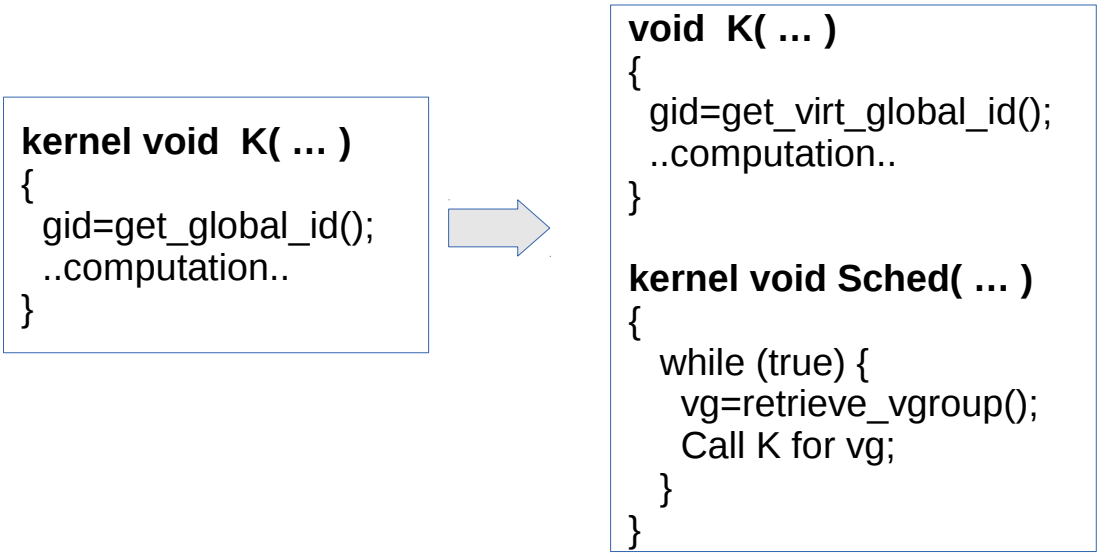


Figure 6.4: A high level schema of our JIT compiler transformation targeting OpenCL Kernels. We convert OpenCL functions to regular functions and we create a new scheduling kernel that accesses a *Virtual Execution Range* via runtime calls, retrieves *virtual groups* and performs their computation by calling the converted function. Our transformation replaces OpenCL work-item functions with runtime calls.

as shown in figure 6.4. It now consists of a simple loop that dynamically dequeues a virtual group and executes it. This means all the original work is done but uses less physical resources. In figure 6.3b, work groups WG0 and WG1 dequeue virtual groups and execute them. The actual virtual groups executed per compute unit will vary due to dynamic scheduling. The reduction of NDrange and the software scheduling of virtual groups ensure fair sharing of accelerator resources and efficient allocation of work to compute units.

The operations of accelOS take place transparently and do not require any modification of application code, or changes in the existing software stack. Furthermore, no hardware changes are required.

6.3 Accelerator Resource Sharing Scheme

The key issue for our fair sharing scheme is determining the right number of work groups per kernel execution. We wish to determine the appropriate number of work groups for each kernel execution so that they all approximately allocate equal resources. We consider modern accelerator architectures with compute units that my

host multiple work group executions at a time if their resource requirements can be satisfied. There are three resources that we need to consider for accelerator sharing: thread number, local memory usage and register usage.

Thread number: We first have to constrain the number of work groups each kernel i executes so that all the concurrent threads can execute concurrently. If T is the maximum number of threads a device can execute, and w_i is the size of a work group for each kernel i then we must constrain the number of work groups x_i for each kernel:

$$\sum_i x_i w_i \leq T$$

To ensure that each kernel has roughly the same resource we have:

$$min_i(min_j(|x_i w_i - x_j w_j|))$$

where we try to minimise the difference in resource between all kernels.

Local Memory: A similar set of constraints can be built for local memory usage. Let L be the maximum local memory available, and m_i is the memory usage of a work group then the number of work groups y_i per kernel is:

$$\sum_i y_i m_i \leq L$$

Again to ensure that each kernel has roughly the same resource we have:

$$min_i(min_j(|y_i m_i - y_j m_j|))$$

Registers: Again a similar set of constraints can be built for register usage. Let R be the maximum registers available, and r_i is the register usage of a work group then the number of work groups z_i per kernel is:

$$\sum_i z_i r_i \leq R$$

Again to ensure that each kernel has roughly the same resource we have:

$$min_i(min_j(|z_i r_i - z_j r_j|))$$

Determining Number of Work Groups: Each of the three constraints can be approximately solved as follows:

$$x_i = \frac{T}{Kw_i}, y_i = \frac{L}{Km_i}, z_i = \frac{R}{Kr_i}$$

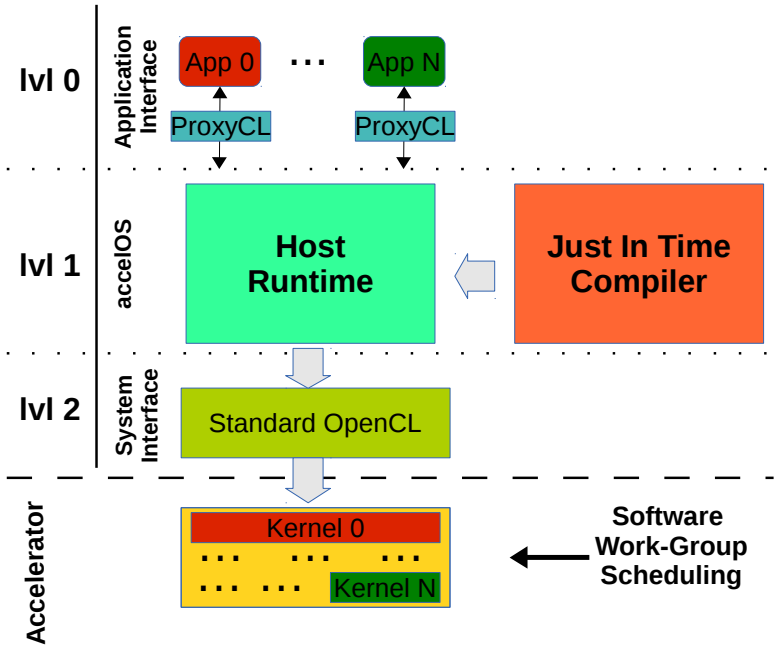


Figure 6.5: accelOS Infrastructure Overview. It is organized in three levels. The application interface (level 0), accelOS core (level 1) and systems interface (level 2). accelOS core consists of two components, the host runtime and the Just In Time (JIT) compiler. The runtime monitors OpenCL applications, manages accelerator resources and schedules kernel execution requests on the accelerator. The JIT compiler transforms kernel codes and links them against a GPU runtime library that enables software work group scheduling.

Given that all constraints must hold simultaneously the final work group size is $min(x_i, y_i, z_i)$. As these are Diophantine equations the resulting work group sizes may be conservative. If not all resources are used, we apply a simple greedy heuristic to incrementally increase the number of work-groups iteratively across the kernel executions until resource saturation.

6.4 Infrastructure Overview

We provide a high level overview of the accelOS infrastructure as it is shown in figure 6.5. accelOS monitors OpenCL applications, manages accelerator resources and enables fair accelerator sharing. Our work remains fully compatible and portable to existing software stacks, accelerator vendors and applications. accelOS infrastructure is seamlessly integrated in existing systems without requiring code modifications or recompilation. Our infrastructure which supports the techniques described above is

organized in 3 levels.

Application interface (level 0): It is responsible for monitoring and interacting with OpenCL applications. The monitoring is done via a library called **ProxyCL** which replaces standard OpenCL. The communication between ProxyCL and accelOS is done over Interprocess Shared Memory[83] which guarantees low overhead. This approach has been proposed in PALMOS[71].

accelOS (level 1): It is a background system process that provides the core functionality of accelOS and consists of two components, a host runtime and a Just in Time (JIT) compiler. The host runtime monitors OpenCL applications via Proxy CL, manages accelerator resources and schedules kernel execution requests. It is further described in section 6.5. The JIT compiler transforms kernel codes and links them against a GPU scheduling library in order to support software work group scheduling. It is further described in section 6.6.

System Interface (level 2): It is the connection of accelOS with the existing system infrastructure. We use standard OpenCL in order to leverage accelerators. This way we can deploy our work on existing systems.

6.5 Host Runtime

This section presents the host runtime of accelOS. It consists of two components described below.

6.5.1 Application Monitor

This is the only component of accelOS that interacts with applications via **ProxyCL**. It monitors OpenCL requests made by applications. If these requests involve new kernel code compilation or kernel execution special actions take place. The finite state machine (FSM) of figure 6.6 presents its operation. When an application performs an OpenCL request three scenarios may take place. (a) If the request creates a new OpenCL Program the JIT compiler takes control, analyzes and transforms the kernel code. The original operation is then performed with the transformed version of the code. (b) If the request is a new kernel execution, Kernel Scheduler takes control, which changes the number of work groups in order to control resource allocation and schedules its execution. (c) For any other request, the application continues its execution instantly and accelOS does not intervene.

6.5.2 Kernel Scheduler

It centrally manages the scheduling of kernel execution requests. It leverages the accelerator resource sharing algorithm described in section 6.3 to select the number of work groups for each kernel execution. For every request, it first constructs a *Virtual Kernel Execution Range* which is copied in the accelerator memory. It then alters the global size of the Kernel Execution Range to match the new number of work groups. It does not modify the work group size or the dimensions of the computation. Finally, it launches the kernel.

The host runtime is built exclusively in user-space and relies on standard POSIX and OpenCL libraries.

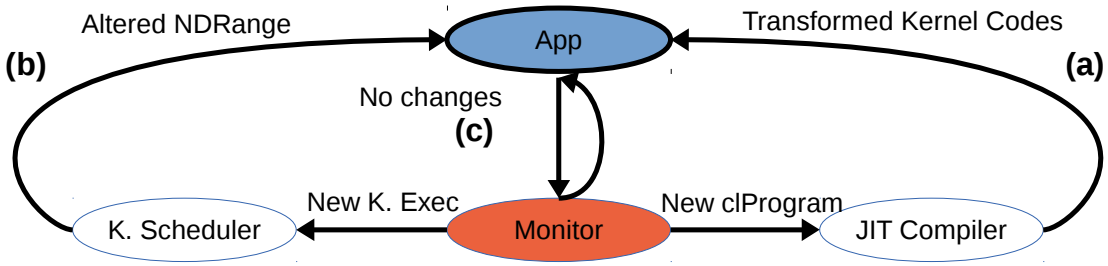


Figure 6.6: Application Monitor Operation. Each Time an application performs an OpenCL call, the monitor component checks its type. Special actions take place if an operation either involves compilation of new kernel code or a kernel execution.

6.6 Just In Time Compilation

Our Just In Time (JIT) compiler intervenes in the standard compilation procedure of OpenCL kernels. It transforms kernel codes and links them statically against a runtime library that enables the software scheduling of virtual groups by accessing a Virtual Execution Range as we described in section 6.2.3. Its operations remain transparent to the application and no modifications are required. Our compiler infrastructure is based on LLVM[63] while we rely on vendor toolchains for target code generation.

This section first describes how we intervene to the standard compilation procedure. We then describe our code transformation technique. Lastly, we present our GPU scheduling library.

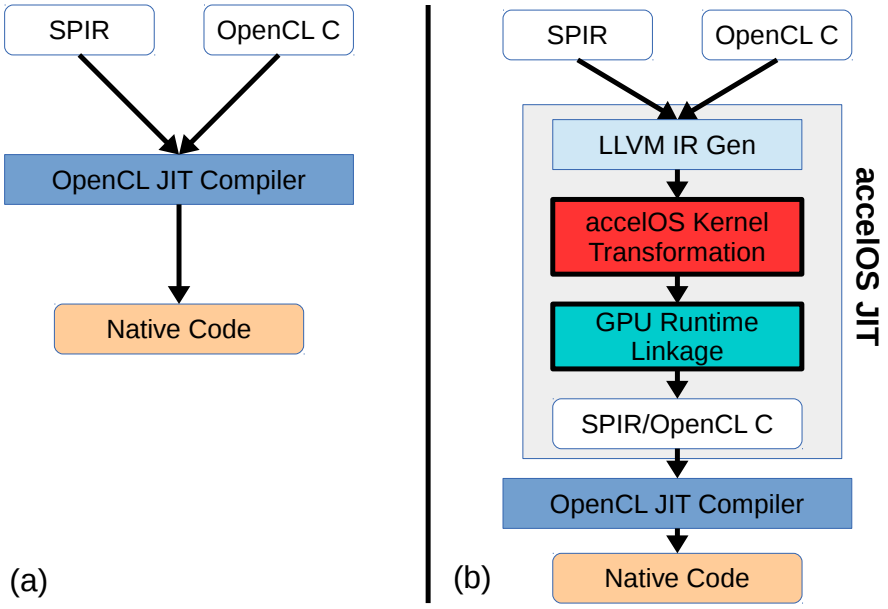


Figure 6.7: OpenCL Kernel Compilation Procedure on (a) Standard OpenCL against (b) accelOS. We intervene the standard compilation procedure, we analyze and transform the kernel code and link it against our GPU scheduling library. We remain trasparent to the existing software stack and no application modification is required.

6.6.1 Compilation Procedure

Figure 6.7a presents the compilation procedure under standard OpenCL. The application provides the kernel code either in OpenCL C or SPIR representation[59]. The vendor compiler then performs a set of optimizations and generates native code for the accelerator.

Figure 6.7b presents our scheme. We intercept the OpenCL call that provides the kernel code. If the code is given in OpenCL C we use Clang to generate LLVM IR. SPIR representation is already compatible with LLVM IR. We instantiate an LLVM Pass Manager and load our compiler passes. We transform kernel codes and statically link them against our GPU scheduling library. Next, if the vendor compiler supports SPIR, we generate SPIR code. Otherwise, we generate OpenCL C. Finally, we use the vendor compiler for the target code generation.

6.6.2 Transformation Overview

Our compiler transformation enables software work group scheduling on existing OpenCL kernels without requiring any change or action from the developer. For every OpenCL

kernel we perform the following:

1. Convert OpenCL function to a regular computation function.
2. Extend the function interface with pointer arguments to the data structures of the runtime.
3. Replace built-in work item functions with runtime equivalents.
4. Create a scheduling kernel function. Its interface includes all the arguments of the original kernel function plus pointer arguments to the runtime data structures.
5. Generate the scheduling kernel body that atomically accesses the Virtual NDRange of the kernel execution and calls the computation function for every virtual group.

The next paragraphs present a kernel transformation example.

6.6.2.1 Kernel Transformation Example

Consider the code of figure 6.8a where each work item either adds or subtracts the input of two buffers depending on its group ID. The kernel function arguments are two input and one output buffers.

Figure 6.8b presents the transformed version of the code. We first convert the kernel function to a regular function and we replace the built-in, work item functions of OpenCL with runtime function calls as it is shown in the lines 5 and 6. The runtime functions require access to data structures and for that reason we have added three trailing arguments to the function interface. The first, **rt**, provides access to the *Virtual Kernel Execution Range (Virtual NDRange)*, the second, **sd**, to scheduling information which is local at work group level. The last, **hdlr**, is a special runtime handler.

6.6.2.2 Software Scheduling Control

Lines 14 to 33 of figure 6.8b consist the code that controls the dynamic scheduling of the virtual groups. We define the scheduling code as a kernel function with the arguments of the original kernel function followed by three additional arguments. **rt** provides access to the Virtual NDRange, **sd** to scheduling information local to the work group. The last one, **lheap**, is a memory buffer allocated in local memory; it is used as a heap for serving memory allocations.

```
1  kernel void mop(global const float *ina,
2      global const float *inb, global float *out)
3  {
4    size_t gid=get_global_id(0);
5    size_t grid=get_group_id(0);
6
7    if(grid<NConstant)
8        out[gid]= ina[gid] + inb[gid];
9    else
10       out[gid]= ina[gid] - inb[gid];
11
12 }
```

(a) Kernel Code Example.

```
1  void mop(global const float *ina,
2  global const float *inb, global float *out,
3  global struct RT *rt, local struct SD *sd, int hdlr)
4  {
5    size_t gid=rt_global_id(rd, hdlr, 0);
6    size_t grid=rt_group_id(rd, hdlr, 0);
7
8    if(grid<NConstant)
9        out[gid]= ina[gid] + inb[gid];
10   else
11       out[gid]= ina[gid] - inb[gid];
12 }
13
14 kernel void dyn_sched(global const float *ina,
15     global const float *inb, global float *out,
16     global struct RT *rt, local struct SD *sd,
17     local void *lheap)
18 {
19   size_t ind;
20
21   if( rt_is_master_work_item() )
22       rt_env_init(rt,&sd);
23
24   for(;;){
25       if( rt_is_master_work_item() )
26           rt_sched_wgroup(rt,&sd);
27       barrier(CLK_LOCAL_MEM_FENCE);
28       if(sd.status==RUN_TERMINATE)
29           break;
30       for(ind=sd.wg_base; ind<sd.wg_end; ++ind)
31           mop(ina, inb, out, r, ind);
32   }
33 }
```

(b) Kernel Code After Transformation.

Figure 6.8: Kernel Code Transformation and Runtime Library Support for software work group scheduling on accelerators. Our compiler transforms the original kernel code. It injects runtime calls and adds control flow for scheduling control and links against our GPU scheduling library.

The scheduling environment is initialized in lines 21 and 22. This call is made by a single work item, the *master* of the work group. All the work items proceed to a loop, where the master, line 26, retrieves virtual groups for execution from the Virtual NDRange. We have an adaptive scheduling scheme that may assign more than one virtual groups for execution at a time. For every group we call the function code, line 32, and the runtime work item functions provide the appropriate group and global ID values.

Local Data Hoisting: OpenCL standard exclusively permits declaration of data in local address space as part of a kernel function body and not regular functions. We convert the original kernel code to a regular function and we need to hoist its local data declarations in the scheduling kernel function body. The allocation of hoisted declarations is done via runtime function calls.

6.6.3 GPU Runtime Library

Our library performs the runtime scheduling of virtual groups provided by Virtual Kernel Execution Ranges (Virtual NDRanges). Every work group has a runtime instance performing virtual group scheduling. The library provides operations for environment control and scheduling. It also provides replacements for the work item functions of OpenCL. The original work groups of a kernel execution are now described by virtual groups and our runtime replacements provide the appropriate values for work item functions at runtime.

6.6.4 Adaptive Scheduling

Our runtime operations involve few mathematical operations that consist negligible overhead. The exceptional case is the scheduling operation that involves an atomic operation. Performing software scheduling on kernels with small number of instructions may expose significant overhead. To avoid this, we support the scheduling of multiple virtual groups at a time. We rely on the following heuristic. If the total number of instructions in LLVM IR is less than 10, a scheduling operation assigns 8 virtual groups to the work group at a time. Respectively, 6 groups for less than 20 instructions, 4 groups if less than 30, 2 groups if less than 40. Otherwise, the scheduling is done with 1 work group at a time.

6.7 Experimental Setup

In this section we describe the platforms, workloads, metrics and methodology we use in our evaluation.

6.7.1 Evaluation Platforms

We evaluate our approach on two distinct heterogeneous platforms with GPUs from different manufacturers. Both platforms have the same host processor: an Intel i7 4770K CPU @ 3.50GHz and 16GB of DDR3 RAM at 1600Mhz. The first platform contains an NVIDIA Tesla K20m[80] GPU; while the second has an AMD R9 295X2[3]. Both systems run Linux with kernel version 3.13. We use the NVIDIA OpenCL platform, version 331.79 and the Accelerated Parallel Processing framework of AMD, version 1445.5.

6.7.2 Workloads

We use all the kernels from the OpenCL version of the Parboil benchmark suite [101]. We consider workloads consisting of **2**, **4** or **8** parallel kernel execution requests. We first evaluate all pairwise combinations of kernels. As there are 25 Parboil kernels, this gives $25 \times 25 = 625$ in total. It is impractical to evaluate all the available combinations for workloads of 4 and 8 requests and we evaluate a subset of them. There are 390265 4-program workload combinations from which we randomly selected 16384. There are 1.5×10^{11} 8-kernel combinations from which we randomly selected 32768. To have robust results, each workload is executed 20 times and the mean execution time is reported.

6.7.3 Comparison to other approaches

We present all results relative to the baseline OpenCL environment provided by NVIDIA and AMD. To provide a broader evaluation, we implemented the the `Elastic Kernels` [85] approach. This work was originally aimed at CUDA and required a port to OpenCL.

6.7.4 Metrics

We evaluate our scheme with respect to fairness and throughput using existing metrics.

Fairness Metrics for Accelerator Sharing: A heterogeneous system is considered fair, if the slowdowns of kernel executions running concurrently on the accelerator resources are the same [32][78][98]. We adopt the metrics proposed in [32].

The Individual Slowdown IS_k of a kernel execution k is:

$$IS_k = \frac{T(s)_k}{T(a)_k}$$

where $T(s)$ is the number of nanoseconds it takes to perform the kernel execution while it shares the accelerator with other executions. $T(a)$ is the number of nanoseconds it takes to perform the execution in isolation. *System unfairness*, U is defined:

$$U = \frac{\max(IS_0, IS_1, \ldots, IS_{N-1})}{\min(IS_0, IS_1, \ldots, IS_{N-1})}$$

Fairness improvement over baseline for either our scheme or elastic kernels is a simple ratio: $\frac{U_{baseline}}{U_X}$, where $U_{Standard}$ and U_X are the system unfairness values for standard OpenCL and either our scheme or elastic kernels, respectively.

Kernel Execution Overlap: The amount of time kernels co-execute is another measure of GPU sharing. *Execution overlap O* is defined as

$$O = \frac{T(c)}{T(t)}$$

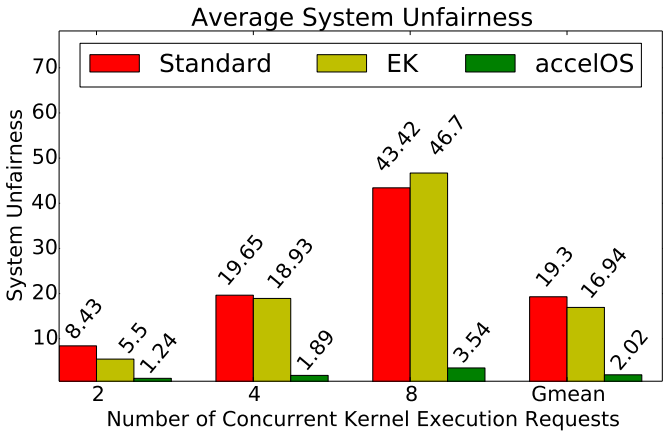
where $T(t)$ is the total time the accelerator is executing at least one of the kernels and $T(c)$ is the amount of time all the kernels are co-executing.

Throughput Speedup: Although we focus on fairness, overall performance is also important. We report overall speedup relative to the baseline i.e. ($\frac{T_{baseline}}{T_X}$) where $T_{baseline}$ is the time for *all* kernels to execute on the standard system and T_x is the time for either our approach or the elastic kernels scheme to execute *all* kernels.

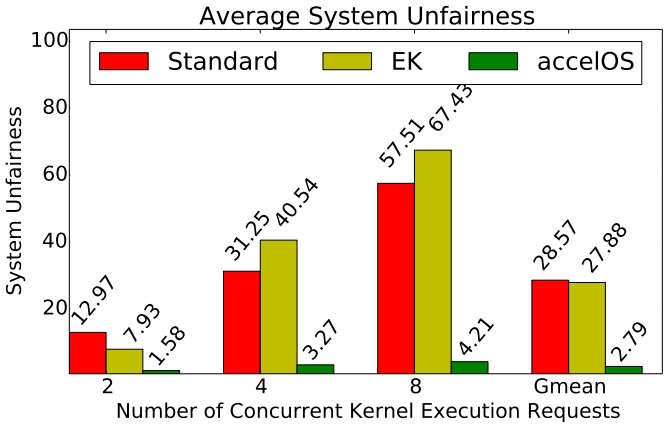
Additional metrics: To evaluate the overhead of our scheme, we measure the time for a single kernel to execute using our approach vs the baseline. We also report average normalized turn around time (ANTT) and worst case ANTT to allow direct comparison with [85]. We also provide STP results [33].

6.8 Results

In this section, we evaluate accelOS which enables resource sharing control on accelerators. accelOS delivers efficient and fair multi-kernel executions, where sets of 2, 4 or 8 parallel kernel execution requests perfrom concurrently.



(a) NVIDIA k20m.



(b) AMD R9 295X2.

Figure 6.9: Average System Unfairness. We compare the system unfairness delived by standard OpenCL and *Elastic Kernels (EK)* against accelOS. Lower is better. accelOS outperforms the other approaches.

We present fairness results in section 6.8.1. Concurrent kernel execution results are discussed in section 6.8.2. Our work has the added bonus of improving system throughput. A detailed description is given in section 6.8.3. We, finally, investigate the performance overhead of accelOS in section 6.8.4. Our optimized version of accelOS compensates the originally introduced overhead and achieves notable performance improvements.

6.8.1 Fairness in Accelerator Sharing

Key goal of this work is to enable resource sharing control on accelerators and enforce fair sharing for multi-kernel executions. Here, we investigate what is the impact of accelOS on fairness. We use the metrics of *Unfairness* and *Fairness Improvement* as

described in section 6.7. For the Unfairness metric, lower values are better while for the Fairness Improvement metric, higher values are better.

6.8.1.1 Result Summary

Figure 6.9a shows the average results on the NVIDIA platform. accelOS reduces unfairness from 8.43 to 1.24 for 2 requests, from 19.65 to 1.89 for 4 requests and from 43.42 to 3.54 for 8 requests. It leads to fairness improvements of 6.8x, 10.4x and 12.27x, while the average improvement is 9.55x. accelOS outperforms *Elastic Kernels (EK)* approach which delivers fairness improvements of 1.53x, 1.03x and 0.93x, respectively and an average improvement of 1.13x.

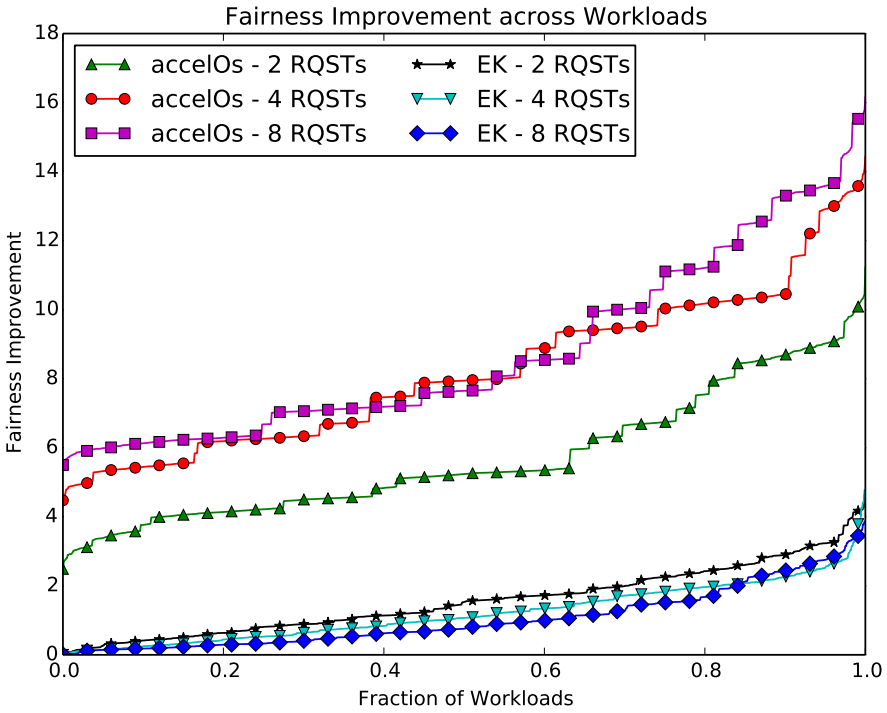
Figure 6.9b shows the results on the AMD platform. Here, the benefits of accelOS are similar to those of NVIDIA. accelOS delivers 8.21x improvement for 2 execution requests, 12.97 against 1.58. In the case of 4 requests, it improves fairness by 9.56x where it reduces unfairness from 31.25 to 3.27. For 8 requests, accelOS improves by 13.66x, reducing unfairness from 28.57 to 2.79. accelOS, again, outperforms *EK* which delivers fairness improvements of 1.63x, 0.77x and 0.85x with an average of 1.02x.

Given the results, accelOS efficiently achieves resource sharing control and leads to significant fairness improvements for multi-kernel executions on both NVIDIA and AMD platforms.

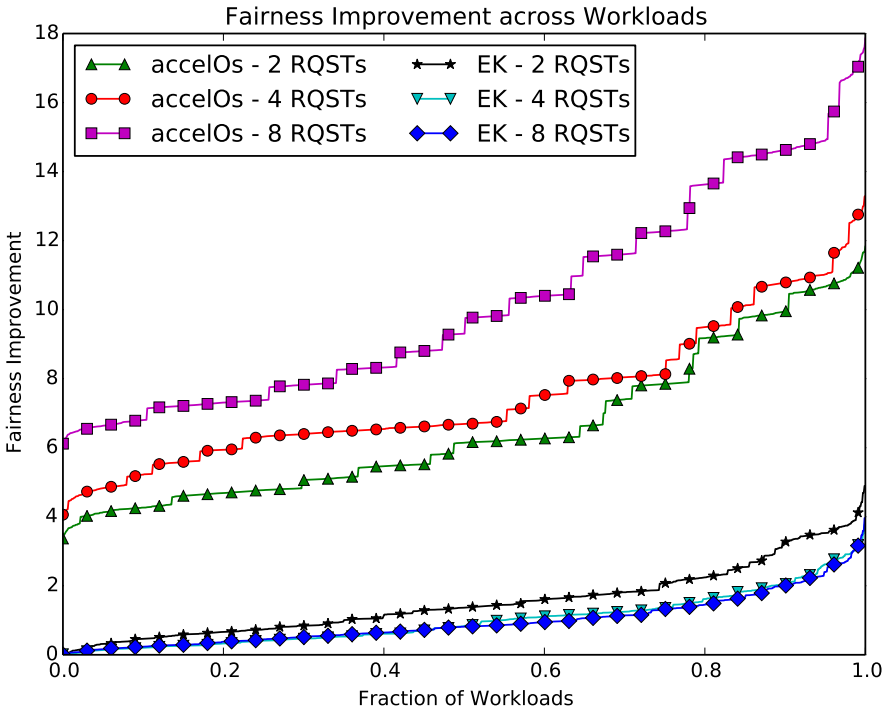
6.8.1.2 Individual Results

Figures 6.10a and 6.10b provide an overview of the fairness improvement results across the workloads we use in our experiments. We provide individual results for workloads of 2, 4 and 8 kernel execution requests on both NVIDIA and AMD platforms for accelOS and *EK*. In case of accelOS, the results range from 0.81x to 15.84x times improvement, where less than 2% of the workloads have a negative fairness result. In contrast, the EK delivers negative results for 44% of the workloads.

accelOS enables dynamic resource sharing control via software managed scheduling on accelerators in contrast to *EK* which relies on static heuristics and static resource allocation. accelOS successfully adapts to large number of requests and fairly assigns system resources while *EK* fails.



(a) NVIDIA K20m.

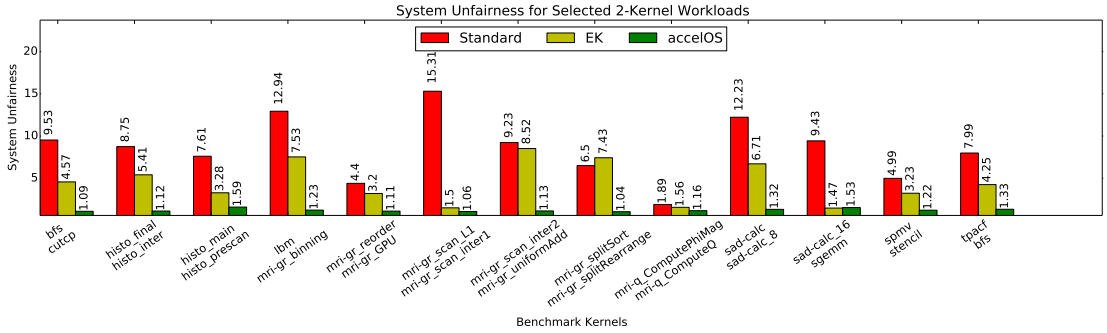


(b) AMD R9 295X2.

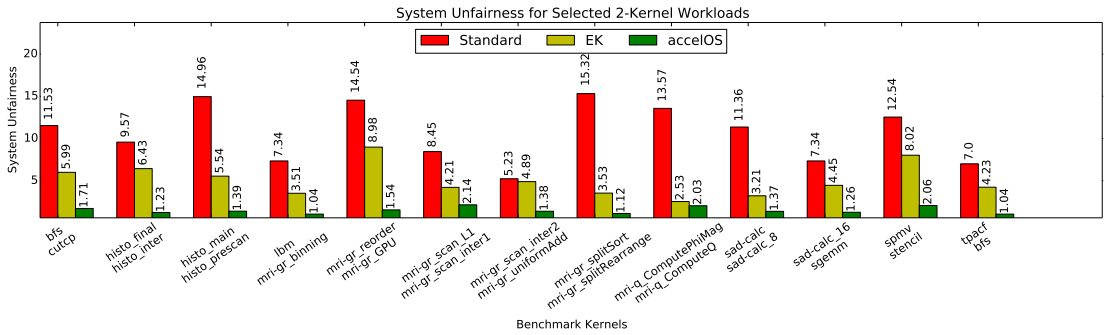
Figure 6.10: Fairness Improvements delivered by accelOS and *Elastic Kernels* for sets of 2, 4 and 8 kernel execution requests. Higher is better. Here, we present the fairness improvement results for all the sets we investigate. The sets are organized in fractions based on their result values. The values increase as we move from left to right. The circle and polygon shapes are meant for easing readability on black and white printouts.

6.8.1.3 Pairwise Results

We provide additional insights on the fairness results delivered by accelOS and *EK* for a selection of 2-kernel workloads in figure 6.11. The selection has been done by pairing the available OpenCL kernels by the alphabetical order of their names. accelOS steadily delivers the best results on both NVIDIA and AMD. We have spotted two cases where *EK* and accelOS deliver nearly the same results. The sad-calc_16 - sgemm pair on NVIDIA and mri-q_computePhiMag - mri-q_ComputeQ pair on AMD suffer from performance degradations due to work group imbalances that negatively affect our software scheduling heuristics. These performance degradations in conjunction with the execution times of these kernels makes accelOS less effective. However, our work delivers significant improvements even for these two cases.

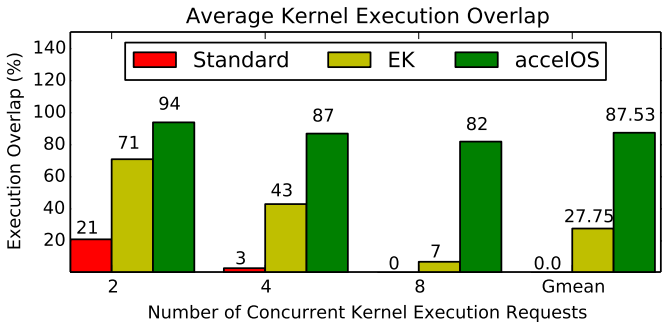


(a) NVIDIA k20m.

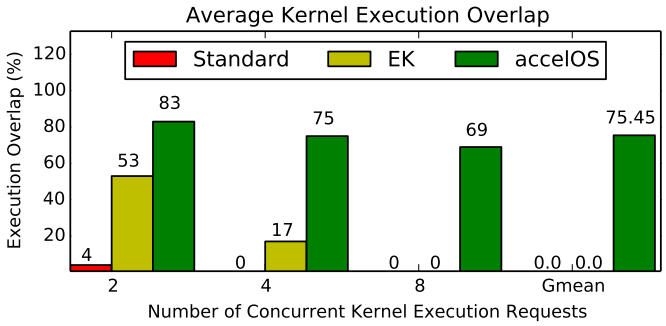


(b) AMD R9 295X2.

Figure 6.11: Unfairness results for a selection of 2-kernel workloads. The selection has been done by pairing the available OpenCL kernels by the alphabetical order of their names. We provide unfairness results for standard OpenCL, Elastc Kernels (EK) and accelOS. Lower is better.



(a) NVIDIA K20m.



(b) AMD R9 295X2.

Figure 6.12: Average Kernel Execution Overlap. Comparison of the kernel execution overlap (percentage) on standard OpenCL and *Elastic Kernels (EK)* against accelOS. Higher is better. accelOS massively improves the percentage of time kernel executions co-exist and perform concurrently on the accelerator.

6.8.2 Concurrent Kernel Executions

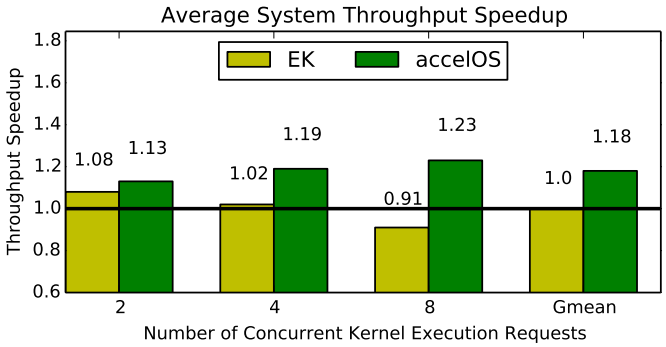
We investigate concurrency for multi-kernel executions. We show that the resource sharing control imposed by accelOS permits multiple kernels to effectively co-exist and compute concurrently. We use the Kernel Execution Overlap metric described in section 6.7. Higher result values are better.

Figure 6.12a provides the results for the NVIDIA platform. In the case of 2 requests, we improve from 21% to 94%. For 4 requests, standard OpenCL delivers 3% while we deliver 87%. Finally, for 8 requests, standard OpenCL delivers 0%, while we enable 82%. accelOS outperforms *Elastic Kernels (EK)* approach which delivers 71%, 43% and 7%, respectively.

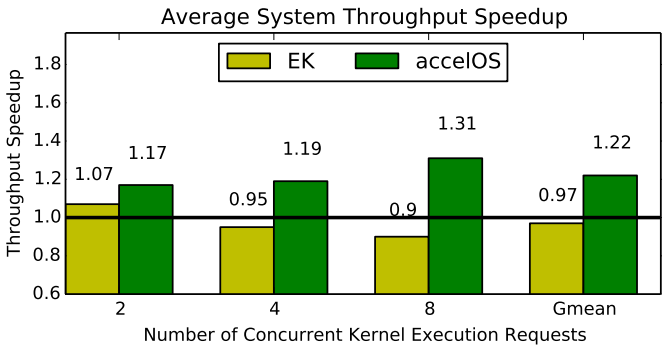
AMD platform behaves worse than NVIDIA. As can be seen in figure 6.12b, standard OpenCL delivers 4%, 0% and 0% for 2, 4 and 8 requests, respectively. accelOS, in contrast, enables improved concurrency. It delivers 83%, 75% and 69% for the 3 request sizes. accelOS again is more efficient than EK which delivers 53%, 17% and

0%, respectively.

Both on NVIDIA and AMD, the Execution Overlap results are lower when we scale up from 2 to 8 requests. This happens because (a) the accelerator multi-tenancy leads to higher resource contention between the kernel executions and (b) the varying kernel workloads that lead to execution time imbalances.



(a) NVIDIA K20m.



(b) AMD R9 295X2.

Figure 6.13: Average System Throughput Speedups for sets of 2, 4 and 8 kernel execution requests. We compare against the *Elastic Kernels (EK)*. The baseline is the standard OpenCL. Higher is better.

6.8.3 System Throughput

We evaluate throughput speedups delivered by accelOS and compare against the *Elastic Kernels(EK)*[85]. The baseline is the standard OpenCL.

6.8.3.1 Result Summary

The results for NVIDIA are shown in figure 6.13a. We deliver an average speedup of 1.13x against 1.08x of *EK* for 2 requests and 1.19x against 1.02x of *EK* for 4 requests.

Finally, we deliver a speedup of 1.23x against 0.91x of *EK* for 8 requests. On average for all the request sizes, accelOS delivers 1.18x while *EK* 1.00x.

The results for AMD are shown in figure 6.13b. We deliver a speedup of 1.17x against 1.07x of *EK* for 2 requests, 1.19x against 0.95x of *EK* for 4 requests. Finally, we deliver a speedup of 1.31x against 0.9x for 8 requests. On average for all the request sizes, accelOS delivers 1.22x while *EK* 0.97x.

accelOS enables resource sharing control and dynamic work group scheduling. This leads to significant throughput results that increase as we scale up to larger number of requests. In contrast, the *EK* approach relies on static heuristics and static resource allocation and fails to manage large number of requests or adapt to dynamic system changes. This is the reason that *EK* delivers negative results for large number of requests.

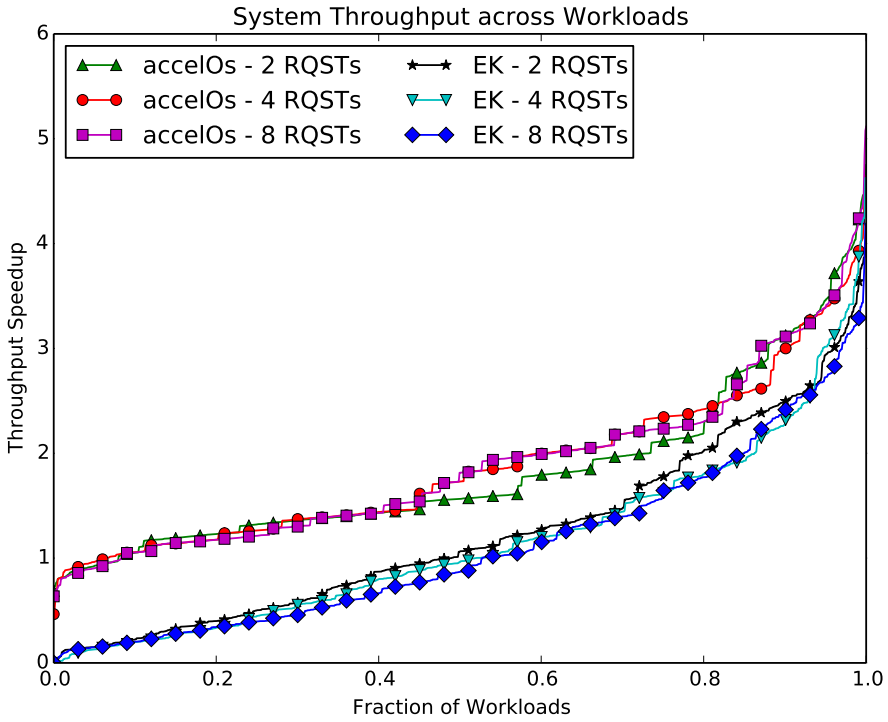
6.8.3.2 Individual Results

Figures 6.14a and 6.14b provide an overview of the throughput speedup results across the workloads we use in our experiments. We provide individual results for workloads of 2, 4 and 8 kernel execution requests on both NVIDIA and AMD platforms for accelOS and *EK*. The throughput speedup results range from 0.52x to 4.8x. Less than 5% of the workloads have slowdowns for accelOS while 54% of the workloads have slowdowns for *EK*.

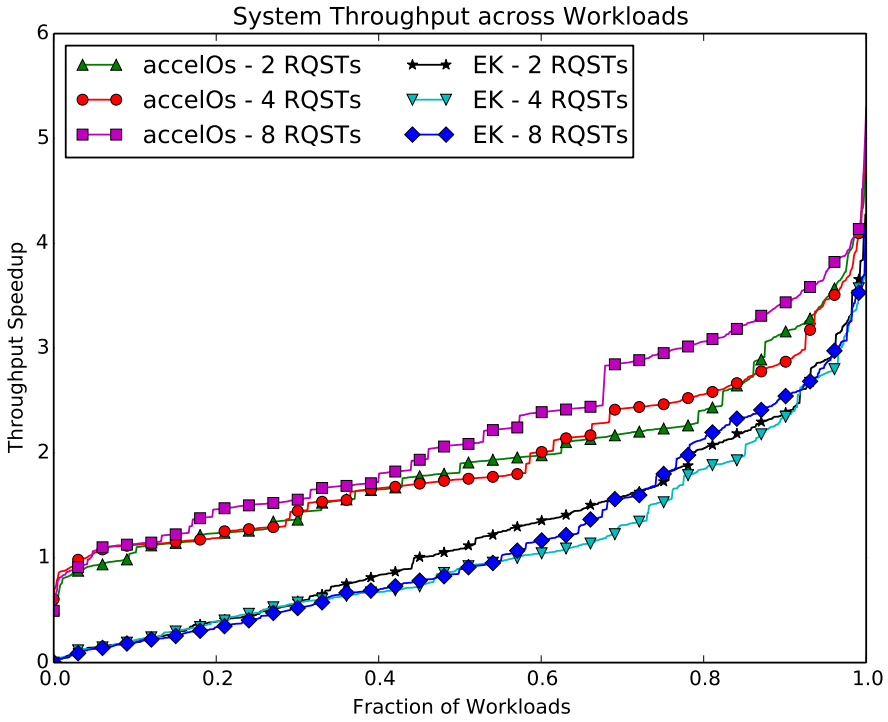
6.8.4 accelOS Overhead

Our infrastructure performs a kernel code transformation and adds a software layer that enables software work group scheduling. These changes raise concerns regarding performance penalties and we investigate them here. We compare kernel execution times delivered by accelOS against Standard OpenCL. We, specially, consider two versions of accelOS, the (a) *naive* and (b) *optimized* versions; the last includes the adaptive scheduling described in section 6.6. Figures 6.15a and 6.15b present the comparison results for the NVIDIA and AMD GPUs, respectively, where we use the speedup metric ($\frac{ExecTime_{Standard}}{ExecTime_{accelOS}}$).

In the case of NVIDIA, shown in figure 6.15a, speedup values range from 0.92x to 1.03x for *naive* and from 0.96x to 1.14x for *optimized*. The geometric average is 0.98x for *naive* and 1.07x for *optimized*. For the *optimized* version, the one we use for our experiments, benchmark kernels sgemm and uniformAdd of mri-gridding have

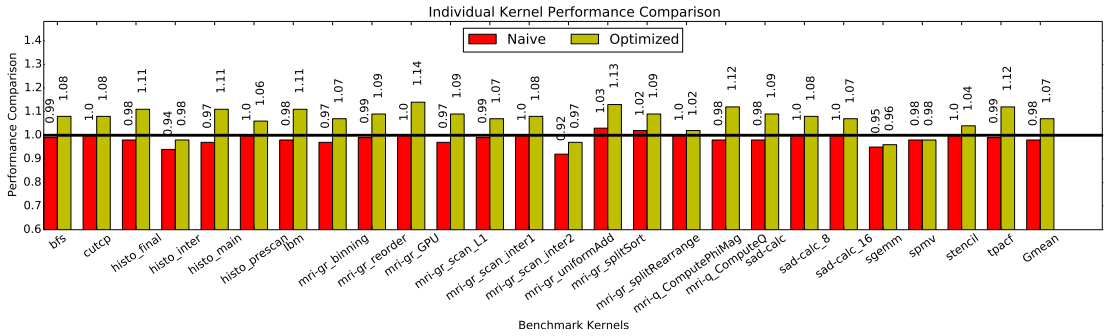


(a) NVIDIA K20m.

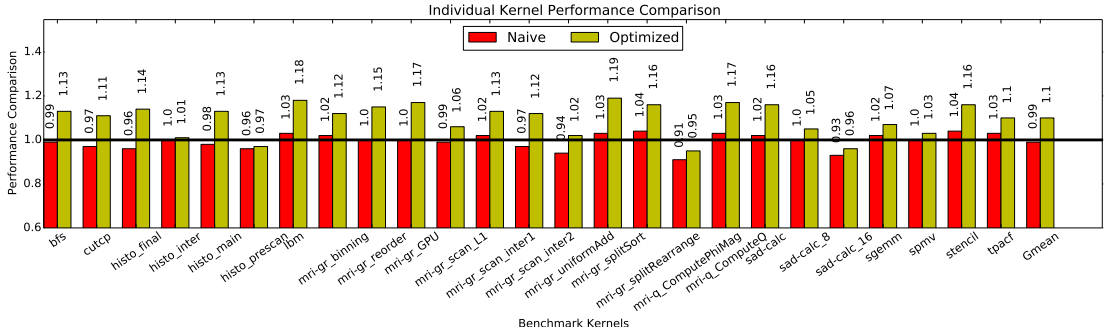


(b) AMD R9 295X2.

Figure 6.14: System Throughput Speedups for sets of 2, 4 and 8 kernel execution requests. We compare against the *Elastic Kernels (EK)*. The baseline is the standard OpenCL. Higher is better. Here, we present the throughput speedup results for all the sets we investigate. The sets are organized in fractions based on their result values. The values increase as we move from left to right. The circle and polygon shapes are meant for easing readability on black and white printouts.



(a) NVIDIA k20m.



(b) AMD R9 295X2.

Figure 6.15: accelOS Performance Impact. We compare accelOS against the standard OpenCL environment. We consider two versions of accelOS, the *naive* and *optimized*. The *naive* leads to small average slowdowns while the *optimized* significantly boosts performance. By default, we use the *optimized* version.

the lowest values, 0.96x and 0.97x, while splitSort of mri-gridding and GPU of mri-gridding have the highest values of 1.13x and 1.14x, respectively.

In the case of AMD, shown in figure 6.15b, speedups range from 0.91x to 1.04x for *naive* and from 0.95x to 1.19x for *optimized*. For the *optimized* version, the one we use for our experiments, kernels such as ComputePhiMag of mri-q and calc_16 of sad have the lowest values, 0.95x and 0.96x, while kernels lbm and splitSort of mri-gridding have the highest values of 1.18x and 1.19x. The geometric average is 0.99x for *naive* and 1.10x for *optimized*.

Our naive implementation of accelOs leads to small slowdowns of 2% and 1%. However our optimized version does not just compensate the overhead but it leads to significant performance gains. This is due to the software work group scheduling which is dynamic and leads to well balanced scheduling. As we describe in section 6.6.4, we consider the overhead of our runtime on short kernels where we use a heuristic to minimize that overhead. However, we still suffer small slowdowns for few

kernels on both platforms.

	EK			accelOS		
# RQSTs	**STP**	**ANTT**	**W. ANTT**	**STP**	**ANTT**	**W. ANTT**
2	1.13	3.57	56.7	1.15	1.12	8.2
4	0.99	4.33	72.2	1.18	1.32	14.2
8	0.93	7.57	87.59	1.25	1.78	23.1

Table 6.1: Additional metrics and measurements comparing accelOS against the *Elastic Kernels (EK)* on NVIDIA K20m. Higher values are better for STP, while lower values are better for ANTT. W. ANTT is the worst ANTT value reported.

	EK			accelOS		
# RQSTs	**STP**	**ANTT**	**W. ANTT**	**STP**	**ANTT**	**W. ANTT**
2	1.04	4.2	64.6	1.18	1.35	13.4
4	0.97	6.83	84.6	1.18	2.12	19.5
8	0.92	7.98	98.54	1.28	3.26	31.34

Table 6.2: Additional metrics and measurements comparing accelOS against the *Elastic Kernels (EK)* on AMD R9 295X2. Higher values are better for STP, while lower values are better for ANTT. W. ANTT is the worst ANTT value reported.

6.8.5 Additional Evaluation Metrics

Prior research work has considered some additional metrics which are STP[85][33] for system throughput evaluation and ANTT[85] as an indirect metric for quantifying system fairness. We provide a brief summary of the average results for Elastic Kernels and accelOS on NVIDIA, in figure 6.1 and on AMD, in figure 6.2. accelOS clearly delivers better results on both platforms.

6.9 Summary

In this chapter we presented accelOS, a runtime and compiler infrastructure that enables software work group scheduling on accelerators. It enables fair accelerator sharing, efficient multi-kernel executions and throughput speedups. accelOS integrates seamlessly with the existing software stack and it does not require any modification or recompilation of the applications, libraries or drivers.

We delivered fairness improvements ranging from 6.8x to 13.66x for multi-kernel

workloads of various sizes. Furthermore, we deliver system throughput speedups ranging from 1.13x to 1.31x.

The last contribution of this thesis has been presented in this chapter. The next chapter concludes the thesis by providing a brief summary of all the contributions as well as a critical analysis and directions for future work.

Chapter 7

Conclusion

This thesis has introduced several methods for addressing key challenges of multi-tasking on heterogeneous systems. A host-device communication optimization that reduces communication overhead has been presented in chapter 4. Chapter 5 presents a heterogeneous accelerator layer that enables central management of accelerators and fine-grained inter-vendor accelerator sharing. A technique and a software stack infrastructure that enable resource sharing control on accelerators is presented in chapter 6. All the contributions presented in this thesis seamlessly integrate to existing software stacks and are portable across vendors. In addition, they do not require any application code modification or recompilation and preserve existing programming models.

The structure of this chapter is organized as follows. Section 7.1 briefly summarizes the contributions of this thesis. A critical analysis of the contributions is given in section 7.2. Future work directions are given in section 7.3.

7.1 Contributions

This section summarizes the main contributions of this thesis as they have been presented in the previous three chapters.

7.1.1 Host-Device Communication Optimization

A technique for host-device communication optimization is presented in chapter 4. It reduces the communication overheads introduced by heterogeneous system designs where additional data transfers are required between distinct physical memories. This

work proposes a technique that relies on automatic platform characterization and application tracing and transparently improves application performance. It does that by selecting the appropriate memory allocation and communication policies. The proposed design does not require any application modification, integrates seamlessly to the existing software stack and is portable across vendors. It delivers significant performance improvements for a large number of benchmarks on three platforms.

7.1.2 Heterogeneous Acceleration Layer

A heterogeneous acceleration layer, named *PALMOS*, is presented in chapter 5. PALMOS is a user-space virtualization layer that enables central management of accelerator resources, fine-grained inter-vendor accelerator sharing and efficient workload scheduling. Its operation remains completely transparent to applications and no application changes are required. It interacts with applications via the OpenCL standard and relies on the programming model of OpenCL. The PALMOS design is portable and supports multiple vendors on a single system configuration. It delivers performance improvements for individual applications and system throughput improvements for multi-application workloads. Multiple system configurations have been evaluated where different number and types of accelerators are available.

7.1.3 Resource Sharing Control on Accelerators

The final contribution of this thesis is a technique and software infrastructure for resource sharing control on accelerators, named *accelOS*, described in detail in chapter 6. It provides a method to manage accelerator sharing for parallel workload execution requests arriving from multiple users and applications. It enforces fair access to accelerator resources and concurrent accelerator sharing. accelOS is seamlessly integrated to existing software stacks and does not require any application modification or changes in accelerator architecture. It dramatically improves system fairness while delivering system throughput improvements and enables multi-tasking on accelerators.

7.2 Critical Analysis

While this thesis has presented some significant contributions to the field of heterogeneous computing, some aspects of the used methods and approaches demand a critical analysis. Four issues are described in the next sections.

7.2.1 Alternative Designs in Kernel Space

In this thesis, all the proposed system designs and software development concern software stack components and compiler infrastructures that operate in user space. There are no contributions at the Operating System level. This is due the limitations of the current software infrastructure. Each accelerator vendor provides its proprietary version of drivers and runtime libraries which do not follow any standard development practice or development interface. OpenCL, which is a user-space library, is the only shared interface across different vendors and accelerator categories. Under the current scheme, research prototypes and production systems that target portability and inter-vendor heterogeneity should rely on the user-space library implementations of OpenCL. Furthermore, many operations required for the management of heterogeneous systems introduce significant complexity and their potential merge in kernel space may introduce performance or resilience issues. However, hybrid designs that involve both operations in user and kernel spaces are definitely worth exploring. There is a quite clear need for inter-vendor interoperability and well defined standards for accelerator management and sharing.

7.2.2 Feedback Driven Resource Management

The proposed contributions perform resource management by monitoring the activity of applications, controlling their operations on accelerators and keeping track of system status. However, there is no adaptive mechanism where our infrastructure improves its resource management decisions by analyzing runtime information. This information may be application performance, resource contention and the interaction of different applications that co-exist on the system. Hardware performance counters provide access to significant information regarding application performance and system behavior. However, current architectures tend to support single application profiling and tracing aimed of later off-line usage of the collected data. Architecture designs with enhanced hardware counters and software support for dynamic multi-application monitoring could be useful for their integration in adaptive schemes. Furthermore, accelerators have limited support for performance counters which typically follow a trade-off of either sacrificing measurement precision or monitoring a subset of accelerator resources. This approach may sufficiently serve traditional profiling needs but introduces significant challenges for their leverage by adaptive resource management schemes.

7.2.3 Unified Management of Computation and Graphics Workloads

This thesis exclusively considers multi-tasking on heterogeneous systems for computation workloads and there is no consideration of graphics processing. All the presented contributions focus on improving multi-tasking and execution of computation workloads. This limitation may not be an issue for multi-tasking systems that are dedicated to computation, such as data center nodes. However, it is a significant problem for mobile devices and desktop systems where applications are a mix of computation and graphics workloads. Current software stack and vendor designs do not provide the required access for dynamically managing accelerator resources for both computation and graphics tasks. However, recently introduced standards, such as Vulkan, enable a unified programming model for computation and graphics operations on accelerators. This new model potentially provides the required abstraction and resource access for investigating unified management of both workload types.

7.2.4 Performance Evaluation with non GPU accelerators

The contributions of this thesis follow a portable design that adapts to different types of heterogeneous systems. However, all the performance evaluation has exclusively been done on heterogeneous systems consisting of CPUs and GPUs of multiple vendors. This is due to the easy access to GPU processors. Evaluation of the presented contributions on systems equipped with non GPU accelerator types such as Xeon Phi and FPGAs could provide more insights for these platforms and potentially explore application areas that are not fully covered by computing on GPUs.

7.3 Future Work

This section briefly introduces directions on how the presented work could be extended in the future.

7.3.1 Unified Management of Computation and Graphics

This work enables efficient multi-tasking for computation workloads on heterogeneous systems while ignoring graphics processing requirements of applications. One potential direction for future work is to extend the designs of this thesis to provide unified

management of computations and graphics on accelerators. New programming standards, such as Vulkan, may support this effort.

7.3.2 Workload Migration across Processors

Modern applications typically consist of multiple execution phases which have different behavior and resource requirements. A scheme that supports the migration of different application phases across processor types can benefit application and system performance while improving power efficiency.

7.3.3 Dynamic Code Optimizations

Modern computing systems comprise diverse processor types including CPU and GPUs of radically different architectures. Each processor architecture benefits by different application code optimizations. Due to the high diversity of existing systems an application cannot be statically optimized for every possible system. A Just In Time compiler infrastructure could transparently perform the required optimizations on the target system during application execution.

7.3.4 Power Aware Resource Management

The contributions of this thesis provide an infrastructure that enables central accelerator management and resource sharing control. This infrastructure could be extended to be aware of power requirements and perform power efficient resource management.

7.3.5 Integrated and Mobile GPUs

The work of this thesis could be extended by specially considering integrated and mobile graphics chips. These architectures provide functionality not found in discrete GPUs, such as memory coherency, which could enhance multi-tasking on heterogeneous systems.

7.3.6 Operating Systems running on Accelerators

Accelerators have evolved to complex processor architectures which serve an increasing number of diverse tasks. Having accelerators with dynamic and adaptive resource allocation, memory management and scheduling is crucially important. This would

be ideally served by an Operating System (OS) running on the accelerator resources. Under this scheme a heterogeneous system would have a full OS running on its host resources and satellite OSes running on each accelerator. The host OS would then periodically exchange coordination messages and directives with the satellite OSes. This approach would require architecture and Operating System modifications but it would provide great flexibility for efficient resource management and software execution.

7.4 Summary

This chapter concludes the thesis. It first provided a summary of the main contributions and then presented a critical analysis of various work aspects. Lastly, this chapter presented directions for future work.

Bibliography

[1] J.T. Adriaens, K. Compton, Nam Sung Kim, and M.J. Schulte. The case for gpgpu spatial multitasking. In *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, pages 1–12, Feb 2012. (Cited on pages 33 and 113.)

[2] Bowen Alpern, Joshua Auerbach, Vasanth Bala, Thomas Frauenhofer, Todd Mummert, and Michael Pigott. Pds: A virtual execution environment for software deployment. In *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, pages 175–185, New York, NY, USA, 2005. ACM. (Cited on pages 38 and 83.)

[3] AMD. Accelerated parallel processing: Opencl programming guide revision 2.7, 2013. (Cited on pages 69, 75, 79, 98, and 128.)

[4] Sara S. Baghsorkhi, Matthieu Delahaye, Sanjay J. Patel, William D. Gropp, and Wen-mei W. Hwu. An adaptive performance modeling tool for gpu architectures. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 105–114, New York, NY, USA, 2010. ACM. (Cited on page 49.)

[5] Gaurav Banga, Peter Druschel, and Jeffrey C Mogul. Resource containers: A new facility for resource management in server systems. In *OSDI*, volume 99, pages 45–58, 1999. (Cited on pages 38 and 83.)

[6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM. (Cited on pages 36 and 83.)

[7] Rajkishore Barik, Rashid Kaleem, Deepak Majeti, Brian T. Lewis, Tatiana Shpeisman, Chunling Hu, Yang Ni, and Ali-Reza Adl-Tabatabai. Efficient mapping of irregular c++ applications to integrated gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 33:33–33:43, New York, NY, USA, 2014. ACM. (Cited on page 41.)

[8] Michael Bauer, Henry Cook, and Brucek Khailany. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 12:1–12:11, New York, NY, USA, 2011. ACM. (Cited on page 47.)

[9] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM. (Cited on page 34.)

[10] M. Becchi and S. Cadambi. Memory-efficient regular expression search using state merging. In *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 1064–1072, May 2007. (Cited on page 65.)

[11] Michela Becchi, Kittisak Sajjapongse, Ian Graves, Adam Procter, Vignesh Ravi, and Srimat Chakradhar. A virtual memory based runtime to support multitenancy in clusters with gpus. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 97–108, New York, NY, USA, 2012. ACM. (Cited on pages 35, 99, and 106.)

[12] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, November 2000. (Cited on pages 44 and 90.)

[13] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference*

on Programming Language Design and Implementation, PLDI '06, pages 158–168, New York, NY, USA, 2006. ACM. (Cited on page 45.)

[14] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 114–124, New York, NY, USA, 2001. ACM. (Cited on page 44.)

[15] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, February 1984. (Cited on page 89.)

[16] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous branch and warp interweaving for sustained gpu performance. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 49–60, Washington, DC, USA, 2012. IEEE Computer Society. (Cited on pages 43 and 113.)

[17] Francisco J Cazorla, Peter MW Knijnenburg, R Sakellariou, E Fernandez, A Ramirez, and M Valero. Qos for high performance smt processors for embedded systems. *IEEE MICRO*, 24(4):24–31, 2004. (Cited on pages 32 and 113.)

[18] Shuai Che, M. Boyer, Jiayuan Meng, D. Tarjan, J.W. Sheaffer, Sang-Ha Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54, Oct 2009. (Cited on pages 28, 49, and 69.)

[19] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: Optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 13:1–13:11, New York, NY, USA, 2011. ACM. (Cited on page 48.)

[20] Jianmin Chen, Xi Tao, Zhen Yang, Jih-Kwon Peir, Xiaoyuan Li, and Shih-Lien Lu. Guided region-based gpu scheduling: Utilizing multi-thread parallelism to hide memory latency. In *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 441–451, May 2013. (Cited on pages 43 and 113.)

[21] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, Eric Roman, Davide B. Bartolini, Nitesh Mor, Krste Asanović, and John D. Kubiatowicz. Tessellation: Refactoring the os around explicit resource containers with continuous adaptation. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 76:1–76:10, New York, NY, USA, 2013. ACM. (Cited on page 38.)

[22] ID Coope. Circle fitting by linear and nonlinear least squares. *Journal of Optimization Theory and Applications*, 76(2):381–388, 1993. (Cited on page 61.)

[23] B.R. Coutinho, G.L.M. Teodoro, R.S. Oliveira, D.O.G. Neto, and R.A.C. Ferreira. Profiling general purpose gpu applications. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD '09. 21st International Symposium on*, pages 11–18, Oct 2009. (Cited on page 48.)

[24] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991. (Cited on pages 26 and 63.)

[25] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic management: A holistic approach to memory placement on numa systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 381–394, New York, NY, USA, 2013. ACM. (Cited on page 46.)

[26] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM. (Cited on page 82.)

[27] Todd Deshane, Zachary Shepherd, J Matthews, Muli Ben-Yehuda, Amit Shah, and Balaji Rao. Quantitative comparison of xen and kvm. *Xen Summit, Boston, MA, USA*, pages 1–2, 2008. (Cited on page 36.)

[28] Docker. Docker Project. `http://www.docker.com/`. (Cited on pages 39 and 83.)

[29] Micah Dowty and Jeremy Sugerman. Gpu virtualization on vmware's hosted i/o architecture. *SIGOPS Oper. Syst. Rev.*, 43(3):73–82, July 2009. (Cited on page 39.)

[30] J. Duato, A.J. Pena, F. Silla, R. Mayo, and E.S. Quintana-Orti. rcuda: Reducing the number of gpu-based accelerators in high performance clusters. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 224–231, June 2010. (Cited on page 35.)

[31] Christophe Dubach, Perry Cheng, Rodric Rabbah, David F. Bacon, and Stephen J. Fink. Compiling a high-level language for gpus: (via language support for architectures and compilers). In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 1–12, New York, NY, USA, 2012. ACM. (Cited on page 41.)

[32] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 335–346, New York, NY, USA, 2010. ACM. (Cited on pages 33, 113, 115, and 129.)

[33] S. Eyerman and L. Eeckhout. System-level performance metrics for multiprogram workloads. *Micro, IEEE*, 28(3):42–53, May 2008. (Cited on pages 129 and 139.)

[34] Stijn Eyerman and Lieven Eeckhout. Probabilistic job symbiosis modeling for smt processor scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 91–102, New York, NY, USA, 2010. ACM. (Cited on pages 32 and 113.)

[35] Wilson W. L. Fung, Ivan Sham, George Yuan, and Tor M. Aamodt. Dynamic warp formation and scheduling for efficient gpu control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on pages 43 and 113.)

[36] R. Gabor, Shlomo Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 149–160, Dec 2006. (Cited on pages 32 and 113.)

[37] Ada Gavrilovska, Sanjay Kumar, Himanshu Raj, Karsten Schwan, Vishakha Gupta, Ripal Nathuji, Radhika Niranjan, Adit Ranadive, and Purav Saraiya. High-performance hypervisor architectures: Virtualization in hpc systems. In *Workshop on System-level Virtualization for HPC (HPCVirt)*. Citeseer, 2007. (Cited on page 37.)

[38] Isaac Gelado, John H. Kelm, Shane Ryoo, Steven S. Lumetta, Nacho Navarro, and Wen-mei W. Hwu. Cuba: An architecture for efficient cpu/co-processor data communication. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 299–308, New York, NY, USA, 2008. ACM. (Cited on page 48.)

[39] Isaac Gelado, John E. Stone, Javier Cabezas, Sanjay Patel, Nacho Navarro, and Wen-mei W. Hwu. An asymmetric distributed shared memory model for heterogeneous parallel systems. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 347–358, New York, NY, USA, 2010. ACM. (Cited on page 41.)

[40] Sanjay Ghemawat et al. Tcmalloc: Thread-caching malloc. *goog-perftools. sourceforge. net/doc/tcmalloc. html*, 2009. (Cited on page 44.)

[41] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDI*, volume 11, pages 24–24, 2011. (Cited on pages 33 and 113.)

[42] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. Libwater: Heterogeneous distributed computing made easy. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 161–172, New York, NY, USA, 2013. ACM. (Cited on page 35.)

[43] Marisabel Guevara, Chris Gregg, Kim Hazelwood, and Kevin Skadron. Enabling task parallelism in the cuda scheduler. In *Workshop on Programming Models for Emerging Architectures*, pages 69–76, 2009. (Cited on pages 34 and 113.)

[44] Abhishek Gupta, Laxmikant V. Kalé, Dejan S. Milojicic, Paolo Faraboschi, Richard Kaufmann, Verdi March, Filippo Gioachin, Chun Hui Suen, and Bu-Sung Lee. Exploring the performance and mapping of hpc applications to platforms in the cloud. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '12, pages 121–122, New York, NY, USA, 2012. ACM. (Cited on page 37.)

[45] Xin Huo, Sriram Krishnamoorthy, and Gagan Agrawal. Efficient scheduling of recursive control flow on gpus. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 409–420, New York, NY, USA, 2013. ACM. (Cited on page 43.)

[46] ISO. C standard. Technical report, 2011. (Cited on pages 63 and 89.)

[47] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. Dynamically managed data for cpu-gpu architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 165–174, New York, NY, USA, 2012. ACM. (Cited on page 47.)

[48] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic cpu-gpu communication management and optimization. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM. (Cited on page 46.)

[49] Raj Jain, Dah-Ming Chiu, and William Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. 1998. (Cited on pages 33 and 113.)

[50] Feng Ji, A.M. Aji, J. Dinan, D. Buntinas, P. Balaji, Wu chun Feng, and Xiaosong Ma. Efficient intranode communication in gpu-accelerated systems. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 1838–1847, May 2012. (Cited on page 47.)

[51] Feng Ji, Heshan Lin, and Xiaosong Ma. Rsvm: A region-based software virtual memory for gpu. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 269–278, Piscataway, NJ, USA, 2013. IEEE Press. (Cited on page 45.)

[52] Qing Jiao, Mian Lu, Huynh Phung Huynh, and T. Mitra. Improving gpgpu energy-efficiency through concurrent kernel execution and dvfs. In *Code Generation and Optimization (CGO), 2015 IEEE/ACM International Symposium on*, pages 1–11, Feb 2015. (Cited on page 113.)

[53] Adwait Jog, Onur Kayiran, Nachiappan Chidambaram Nachiappan, Asit K. Mishra, Mahmut T. Kandemir, Onur Mutlu, Ravishankar Iyer, and Chita R. Das. Owl: Cooperative thread array aware scheduling techniques for improving gpgpu performance. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 395–406, New York, NY, USA, 2013. ACM. (Cited on page 43.)

[54] Patryk Kaminski. Numa aware heap memory manager. *AMD Developer Central*, 2009. (Cited on page 46.)

[55] Poul-Henning Kamp. malloc (3) revisited. In *USENIX Annual Technical Conference*, 1998. (Cited on page 45.)

[56] Shinpei Kato, Karthik Lakshmanan, Raj Rajkumar, and Yutaka Ishikawa. Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30, 2011. (Cited on pages 34 and 113.)

[57] Shinpei Kato, Michael McThrow, Carlos Maltzahn, and Scott A Brandt. Gdev: First-class gpu resource management in the operating system. In *USENIX Annual Technical Conference*, pages 401–412, 2012. (Cited on page 40.)

[58] Khronos Group. The opencl specification, version 1.2, 2011. (Cited on pages 1, 18, 19, 20, 52, 82, and 112.)

[59] Khronos Group. The spir (standard portable intermediate representation) specification, version 1.2, 2012. (Cited on pages 21, 26, and 124.)

[60] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in opencl for multiple gpus. In *Proceedings of*

the 16th ACM Symposium on Principles and Practice of Parallel Programming, PPoPP '11, pages 277–288, New York, NY, USA, 2011. ACM. (Cited on page 39.)

[61] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007. (Cited on pages 36 and 83.)

[62] Richard P. Larowe, Jr. and Carla Schlatter Ellis. Experimental comparison of memory management policies for numa multiprocessors. *ACM Trans. Comput. Syst.*, 9(4):319–363, November 1991. (Cited on page 46.)

[63] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86, March 2004. (Cited on pages 25 and 123.)

[64] Che-Rung Lee, Shih-Hsiang Lo, Nan-Hsi Chen, Yeh-Ching Chung, and I-Hsin Chung. Gpu performance enhancement via communication cost reduction: Case studies of radix sort and wsn relay node placement problem. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 132–139, May 2012. (Cited on page 47.)

[65] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent cpu-gpu collaboration for data-parallel kernels on heterogeneous systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press. (Cited on page 41.)

[66] Min Lee, A. S. Krishnakumar, P. Krishnan, Navjot Singh, and Shalini Yajnik. Supporting soft real-time tasks in the xen hypervisor. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '10, pages 97–108, New York, NY, USA, 2010. ACM. (Cited on page 37.)

[67] Linux Documentation. Linux programmer's manual, 2014. (Cited on page 87.)

[68] Allen D. Malony, Scott Biersdorff, Wyatt Spear, and Shangkar Mayanglambam. An experimental approach to performance measurement of heterogeneous parallel applications using cuda. In *Proceedings of the 24th ACM International*

Conference on Supercomputing, ICS '10, pages 127–136, New York, NY, USA, 2010. ACM. (Cited on page 48.)

[69] Christos Margiolas and Michael F. P. O'Boyle. Portable and transparent host-device communication optimization for gpgpu environments. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 55:55–55:65, New York, NY, USA, 2014. ACM. (Cited on page 5.)

[70] Christos Margiolas and Michael F.P. O'Boyle. accelos: Portable and transparent software managed scheduling on accelerators for fair resource sharing. Under Submission. (Cited on page 5.)

[71] Christos Margiolas and Michael F.P. O'Boyle. Palmos: A transparent, multi-tasking acceleration layer for parallel heterogeneous systems. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 307–318, New York, NY, USA, 2015. ACM. (Cited on pages 5, 113, and 122.)

[72] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 619–630, New York, NY, USA, 2013. ACM. (Cited on page 82.)

[73] C. McCurdy and J. Vetter. Memphis: Finding and fixing numa-related performance problems on multi-core platforms. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 87–96, March 2010. (Cited on pages 49 and 96.)

[74] Paul B Menage. Adding generic process containers to the linux kernel. volume 2 of *Linux Symposium*, pages 45–58, 2007. (Cited on page 84.)

[75] Konstantinos Menychtas, Kai Shen, and Michael L. Scott. Disengaged scheduling for fair, protected access to fast computational accelerators. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 301–316, New York, NY, USA, 2014. ACM. (Cited on page 34.)

[76] Message Passing Interface Forum. Mpi: A message-passing interface standard, version 3.0. Technical report, 2012. (Cited on pages 23 and 89.)

[77] Paulius Micikevicious. Multi-gpu programming. GTC 2012. (Cited on page 96.)

[78] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 146–160, Washington, DC, USA, 2007. IEEE Computer Society. (Cited on page 129.)

[79] Veynu Narasiman, Michael Shebanow, Chang Joo Lee, Rustam Miftakhutdinov, Onur Mutlu, and Yale N. Patt. Improving gpu performance via large warps and two-level warp scheduling. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 308–317, New York, NY, USA, 2011. ACM. (Cited on pages 43 and 113.)

[80] NVIDIA. Nvidia kepler gk110 architecture, 2012. (Cited on pages 69, 83, 98, and 128.)

[81] NVIDIA. *CUDA C Programming Guide, Version 6.0*. 2014. (Cited on pages 1, 11, 52, and 82.)

[82] NVIDIA GRID. Nvidia. `http://www.nvidia.co.uk/object/grid-vdi-desktop-virtualisation-uk.html`. (Cited on page 83.)

[83] Open Group and IEEE. The posix specification, version 1-2008, 2008. (Cited on pages 86, 87, and 122.)

[84] OpenVZ. The openvz linux containers. `http://www.openvz.org/`. (Cited on pages 39 and 83.)

[85] Sreepathi Pai, Matthew J. Thazhuthaveetil, and R. Govindarajan. Improving gpgpu concurrency with elastic kernels. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 407–418, New York, NY, USA, 2013. ACM. (Cited on pages 33, 99, 113, 114, 115, 128, 129, 135, and 139.)

[86] D.C. Pham, T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P.M. Harvey, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D.L. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a

first-generation cell processor. *Solid-State Circuits, IEEE Journal of*, 41(1):179–196, Jan 2006. (Cited on page 1.)

[87] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 431–444, New York, NY, USA, 2013. ACM. (Cited on page 42.)

[88] Rezaur Rahman. *Intel® Xeon Phi Coprocessor Architecture and Tools: The Guide for Application Developers*. Apress, 2013. (Cited on pages 1 and 8.)

[89] Timothy G. Rogers, Mike O'Connor, and Tor M. Aamodt. Cache-conscious wavefront scheduling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 72–83, Washington, DC, USA, 2012. IEEE Computer Society. (Cited on pages 43 and 113.)

[90] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 12–27, New York, NY, USA, 1988. ACM. (Cited on page 26.)

[91] Christopher J. Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. Ptask: Operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 233–248, New York, NY, USA, 2011. ACM. (Cited on page 113.)

[92] Sangmin Seo, Junghyun Kim, and Jaejin Lee. Sfmalloc: A lock-free and mostly synchronization-free dynamic memory allocator for manycores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 253–263, Oct 2011. (Cited on page 45.)

[93] Zhiyong Shan, Xin Wang, Tzi-cker Chiueh, and Xiaofeng Meng. Facilitating inter-application interactions for os-level virtualization. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 75–86, New York, NY, USA, 2012. ACM. (Cited on page 39.)

[94] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: Elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM. (Cited on page 37.)

[95] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. vcuda: Gpu-accelerated high-performance computing in virtual machines. *Computers, IEEE Transactions on*, 61(6):804–816, June 2012. (Cited on page 40.)

[96] Baojiang Shou, Xionghui Hou, and Li Chen. A compiler-assisted runtime-prefetching scheme for heterogenous platforms. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 215–215, Oct 2011. (Cited on page 42.)

[97] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. Gpufs: Integrating a file system with gpus. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 485–498, New York, NY, USA, 2013. ACM. (Cited on page 34.)

[98] Allan Snavely and Dean M. Tullsen. Symbiotic jobscheduling for a simultaneous mutlithreading processor. *SIGPLAN Not.*, 35(11):234–244, November 2000. (Cited on pages 32, 113, and 129.)

[99] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 275–287, New York, NY, USA, 2007. ACM. (Cited on pages 38 and 83.)

[100] Kyle Spafford, Jeremy S. Meredith, and Jeffrey S. Vetter. Quantifying numa and contention effects in multi-gpu systems. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, GPGPU-4, pages 11:1–11:7, New York, NY, USA, 2011. ACM. (Cited on pages 49 and 96.)

[101] John A Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Geng Daniel Liu, and Wen-Mei W Hwu. Parboil:

A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012. (Cited on pages 28, 49, 69, 98, and 128.)

[102] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track*, pages 1–14, 2001. (Cited on page 37.)

[103] Kun Tian, Yaozu Dong, and David Cowperthwaite. A full gpu virtualization solution with mediated pass-through. In *Proc. USENIX ATC*, 2014. (Cited on page 40.)

[104] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *ACM SIGOPS Operating Systems Review*, volume 23, pages 159–166. ACM, 1989. (Cited on pages 32 and 113.)

[105] A. Udupa, R. Govindarajan, and Matthew J. Thazhuthaveetil. Software pipelined execution of stream programs on gpus. In *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*, pages 200–209, March 2009. (Cited on page 41.)

[106] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K Panda. Mvapich2-gpu: optimized gpu to gpu communication for infiniband clusters. *Computer Science-Research and Development*, 26(3-4):257–266, 2011. (Cited on page 47.)

[107] Zhikui Wang, Xiaoyun Zhu, P. Padala, and S. Singhal. Capacity and performance overhead in dynamic resource allocation to virtual containers. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 149–158, May 2007. (Cited on page 38.)

[108] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the denali isolation kernel. *SIGOPS Oper. Syst. Rev.*, 36(SI):195–209, December 2002. (Cited on pages 38 and 83.)

[109] JB White III and Jack J Dongarra. Overlapping computation and communication for advection on hybrid parallel computers. In *Parallel & Distributed*

Processing Symposium (IPDPS), 2011 IEEE International, pages 59–67. IEEE, 2011. (Cited on page 47.)

[110] C.M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi gf100 gpu architecture. *Micro, IEEE*, 31(2):50–59, March 2011. (Cited on page 69.)

[111] Shucai Xiao, P. Balaji, Qian Zhu, R. Thakur, S. Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu chun Feng. Vocl: An optimized environment for transparent virtualization of graphics processing units. In *Innovative Parallel Computing (InPar), 2012*, pages 1–12, May 2012. (Cited on page 35.)

[112] Chao-Tung Yang, Hsien-Yi Wang, Wei-Shen Ou, Yu-Tso Liu, and Ching-Hsien Hsu. On implementation of gpu virtualization using pci pass-through. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 711–716, Dec 2012. (Cited on page 40.)

[113] Yi Yang, Ping Xiang, Jingfei Kong, and Huiyang Zhou. A gpgpu compiler for memory optimization and parallelism management. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 86–97, New York, NY, USA, 2010. ACM. (Cited on page 42.)

[114] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 129–142, New York, NY, USA, 2010. ACM. (Cited on pages 32 and 113.)